Klaus Schneider

The Synchronous Programming Language Quartz

A Model-Based Approach to the Synthesis of Hardware-Software Systems

November 13, 2010, Version 2.0

Department of Computer Science
University of Kaiserslautern
## Contents

1 Introduction ................................................ 1
   1.1 Embedded System Design ............................... 1
   1.2 Models of Concurrent Computation .................... 4
   1.3 Model-based Design Methods ......................... 12
      1.3.1 High-level Synthesis ............................. 12
      1.3.2 Software Synthesis for Embedded Systems ....... 12
      1.3.3 Latency-Insensitive Implementations ............ 12

2 Data Types, Expressions, and Specifications .............. 13
   2.1 Data Types .............................................. 14
      2.1.1 Syntax and Semantics of Data Types ............. 14
      2.1.2 Matching Types to Expected Supertypes .......... 16
      2.1.3 Primitive Recursive Data Types ................. 17
   2.2 Expressions .............................................. 17
      2.2.1 Variables ........................................... 18
      2.2.2 Literals ............................................ 19
      2.2.3 Operators .......................................... 21
      2.2.4 Static (i.e. Compile-time Constant) Expressions . 32
      2.2.5 User-Defined Functions ......................... 33
   2.3 Specifications ............................................ 33
      2.3.1 Syntax ................................................ 34
      2.3.2 Type Rules ........................................... 37
      2.3.3 Semantics ........................................... 39

3 Statements, Interfaces and Modules ........................ 43
   3.1 Modules and Interface Declarations .................. 45
      3.1.1 Interface Declarations ............................. 45
      3.1.2 Statements ........................................... 49
   3.2 Semantic Problems ....................................... 58
      3.2.1 Schizophrenic Statements ......................... 58
      3.2.2 Causality ............................................ 61
VI Contents

3.3 Core Statements ............................................................ 67
3.4 Macro Statements ............................................................ 69
  3.4.1 Simple Macro Statements ............................................. 69
  3.4.2 Additional Wait-Statements .......................................... 70
  3.4.3 Additional Loops ...................................................... 70
  3.4.4 Additional During-Statements ...................................... 70
  3.4.5 Let-Abbreviations .................................................... 70
  3.4.6 Conjunctively Active Parallel Execution ......................... 71
  3.4.7 Generic Sequence and Parallel Statements ....................... 71
  3.4.8 Delay Statements .................................................... 72
  3.4.9 Abstractions .......................................................... 72
  3.4.10 Nondeterminism and Asynchronous Concurrency ................ 73
  3.4.11 Exception Handling ................................................ 75
  3.4.12 Micro Step Variables .............................................. 81
  3.4.13 Inout Variables ..................................................... 81

4 Formal Semantics ........................................................... 83
  4.1 Evaluating Expressions in Incomplete Environments ................ 86
  4.2 SOS Transition Rules ................................................... 88
  4.3 SOS Reaction Rules (Causality Analysis) ............................. 97
  4.4 Complete Operational Semantics ...................................... 102
  4.5 Symbolic SOS Transition Rules: The Control Flow Predicates ... 106
    4.5.1 Computing the Control Flow ..................................... 108
    4.5.2 Computing Guarded Actions ..................................... 115
    4.5.3 Computing the Data Flow ....................................... 124
    4.5.4 Haltset Encoding of Symbolic SOS Rules ....................... 127
  4.6 Symbolic SOS Reaction Rules ......................................... 130
    4.6.1 Causality Analysis with Higher Data Types .................... 130
    4.6.2 Can- and Must-Guarded Actions ................................ 139

5 Compilation to Guarded Actions ......................................... 145
  5.1 Revisiting Schizophrenia Problems .................................. 147
  5.2 Translation to Guarded Actions ...................................... 150
    5.2.1 Computing Guarded Actions of the Surface .................... 151
    5.2.2 Computing Guarded Actions of the Depth ..................... 161
  5.3 Overall Compilation and Linking ................................... 170

6 Synthesis of Synchronous Systems ..................................... 179
  6.1 Synthesis of Data Flow Equations ................................... 179
    6.1.1 Equation Systems as Hardware Circuits ....................... 184
  6.2 Software Synthesis based on Explicit Extended Finite State
       Machines .................................................................... 188
  6.3 Software Synthesis based on Equation Systems .................... 188
    6.3.1 Dataflow Evaluation of Equation Systems ..................... 189
    6.3.2 Example Dataflow Evaluations ................................ 190
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.3</td>
<td>Dataflow Evaluation and Multithreaded Execution</td>
<td>190</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Dataflow Evaluation and Constructive Programs</td>
<td>191</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Adjusting the Granularity by Macro Nodes</td>
<td>194</td>
</tr>
<tr>
<td>6.4</td>
<td>Compiling Synchronous Programs into Graph Code</td>
<td>195</td>
</tr>
<tr>
<td>6.5</td>
<td>Compiling Synchronous Programs into Jobs</td>
<td>195</td>
</tr>
<tr>
<td>6.6</td>
<td>Formal Semantics of the Job Code</td>
<td>197</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Approximating the Environment</td>
<td>198</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Preparing the Next Macro Step</td>
<td>199</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>Synthesis of Asynchronous Systems</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Dataflow Process Networks</td>
<td>215</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Syntax of DPNs</td>
<td>215</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Denotational Semantics of DPNs</td>
<td>219</td>
</tr>
<tr>
<td>7.1.3</td>
<td>DPNs as Rewrite Systems</td>
<td>223</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Operational Semantics of DPNs</td>
<td>230</td>
</tr>
<tr>
<td>7.1.5</td>
<td>The Full Abstraction Problem</td>
<td>238</td>
</tr>
<tr>
<td>7.1.6</td>
<td>The Kahn Principle</td>
<td>241</td>
</tr>
<tr>
<td>7.2</td>
<td>Important Classes of Dataflow Process Networks</td>
<td>241</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Single-Rate Dataflow Process Networks</td>
<td>241</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Synchronous Dataflow Process Networks</td>
<td>242</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Cyclo-Static Dataflow Process Networks</td>
<td>248</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Boolean Dataflow Process Networks</td>
<td>251</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Tagged Dataflow Process Networks</td>
<td>251</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7.3</th>
<th>Globally Asynchronous Locally Synchronous Systems</th>
<th>251</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.1</td>
<td>Synchronous Programs vs. Dataflow Programs</td>
<td>251</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Endochrony and Isochrony</td>
<td>259</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Weak Endochrony</td>
<td>286</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Elastic Synchronous Systems</td>
<td>293</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>Sanity Checks</th>
<th>295</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Surface-Depth Expansion of Statements</td>
<td>295</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Surface/Depth-Splitting without goto</td>
<td>298</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Surface/Depth-Splitting with goto</td>
<td>309</td>
</tr>
<tr>
<td>8.2</td>
<td>Formal Verification of Properties</td>
<td>316</td>
</tr>
<tr>
<td>8.2.1</td>
<td>From Assumptions and Assertions to Specifications</td>
<td>316</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Model Checking of Temporal Logics</td>
<td>316</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Supervisory Control (Reactive Synthesis)</td>
<td>316</td>
</tr>
</tbody>
</table>

| 8.3 | Checking Absence of Overflows | 316 |
| 8.4 | Checking Absence of Write Conflicts | 316 |
| 8.4.1 | Deadend States and Write Conflicts | 317 |
| 8.4.2 | Detection of Write Conflicts | 324 |
# Contents

9 **Hardware-Software Codesign** ........................................... 327
  9.1 Introduction .......................................................... 327
  9.2 Application-Specific Microprocessors ............................. 327
  9.3 Dynamically Reconfigurable Processing Elements .......... 327
  9.4 HW/SW-Codesign based on Virtual Platforms ............... 327

A **Partial Orders** .......................................................... 329
  A.1 Complete Partial Orders ............................................. 329
    A.1.1 Partial Orders and Directed Subsets ......................... 329
    A.1.2 Complete Partial Orders ...................................... 333
    A.1.3 Fixpoint Theorems ............................................. 335
    A.1.4 Cartesian Products of Partial Orders ....................... 340
  A.2 Complete Lattices .................................................... 343
    A.2.1 Lattices ................................................................ 343
    A.2.2 Complete Lattices .............................................. 345
  A.3 Theory of Streams ....................................................... 348
    A.3.1 Complete Partial Order of Streams ......................... 348
    A.3.2 Metric Space of Streams ...................................... 352

B **Computer Arithmetic** .................................................. 357
  B.1 Natural Numbers .......................................................... 358
  B.2 Radix-\(B\) Numbers ..................................................... 362
  B.3 The Integers ............................................................... 389
  B.4 \(B\)-Complement Numbers ............................................ 391
  B.5 Hardware-Implementations ............................................ 409
    B.5.1 Radix-2 Arithmetic .............................................. 409
    B.5.2 2-Complement Arithmetic ..................................... 415
  B.6 Further Representations of Numbers ................................ 417
    B.6.1 Gray Code Numbers .............................................. 419
    B.6.2 Residue Number Systems ...................................... 419
    B.6.3 Signed Digit Numbers ......................................... 419

C **Concrete Grammar of the Quartz Language** ...................... 421
  C.1 Lexical Tokens ........................................................... 421
  C.2 Grammar of the Quartz Language .................................. 423
  C.3 Some Example Programs .............................................. 427

References ................................................................. 429

Index ................................................................. 445
1

Introduction

1.1 Embedded System Design

During the past decades, an incredible change of technology has been observed in many devices: traditionally used mechanical parts have often been first replaced by analog electronic devices, then by digital hardware, and later on by programmable microprocessors. Finally, entire computer systems consisting of a microprocessor with a main memory and specific input/output facilities running some application-specific software have been integrated which lead to the definition of **embedded systems**. One of the first embedded systems was probably the Apollo guidance computer [187] for navigation and control of the Apollo spacecrafts designed in the 1960s. They were considered the most risky parts of the Apollo project.

The use of embedded systems has many advantages: it allowed not only a cheaper production, it also allowed the miniaturization of devices and the integration of more intelligent functionalities. The increased use of software makes embedded system more flexible so that late changes in their design are manageable and even changes after shipping these systems are possible.

Today, the progress in technology is still unbroken, and there is now a trend towards the integration of more processors on a single chip, so that even **heterogeneous multiprocessor systems on a single chip (MPSoCs)** can be used for the design of embedded systems. The combination of a dynamically scheduled general purpose RISC CPU with a statically scheduled VLIW/DSP processor is often viewed classic in some application areas. The use of MPSoCs often allows the tenfold increase in computing power with a tenfold decrease of energy consumption. Huge markets like those for mobile phones or other consumer electronics lead to mass production that made prices for simple microprocessors about one dollar possible. Other important application areas of embedded systems are avionics and automotive industries (e.g. anti-lock braking system (ABS), electronic stability control (ESC/ESP), traction control (TCS) and automatic four-wheel drive), consumer electronics, CD/DVD/BlueRay players, domestic appliances, communication devices, traf-
fic control systems including navigation devices, controllers for industrial plants, and special devices used in medical equipments (pacemakers, PET, CT).

Unfortunately, the design of embedded systems is still not well developed. One problem is that many different disciplines like software engineering, hardware design, control theory as well as mechanical and electrical engineering are involved that have not yet been integrated into a seamless design flow. Another problem is that even new solutions within these disciplines have to be developed to address the special needs of embedded systems. To discuss these issues in software and hardware synthesis for embedded systems that lead to the motivation of new programming languages like synchronous languages, we have to consider the general architecture of embedded systems.

![Fig. 1.1. General Architecture of an Embedded System](image)

The general architecture of an embedded system is shown in Figure 1.1: The embedded system itself may consist of many subsystems each consisting of multiple processors running application-specific software. These computer systems communicate not only with themselves but also directly with their environment. From the technological side, analog-digital converters used in special sensors and actor may be used for the communication between the
environment and the embedded system, and special bus systems \cite{220} like CAN, FlexRay, MOST, LIN are used for the communication of the embedded systems with each other. The communication of single subsystems within one embedded system are supported by special busses like ARM’s AMBA bus. Each of the embedded system consists of software and hardware, where the hardware is furthermore divided into a part consisting of standard components such as micro-processors and special components that are implemented for a particular application. A new emerging trend is also the use of application-specific processors whose instruction sets are adapted to a particular application. Hence, being application-specific computer systems is already a characterizing property of embedded systems.

The communication of the embedded systems with their environment leads to further properties. Unlike classic (transformational) software programs that read inputs at starting time and produce outputs at termination time, many embedded systems are so-called reactive systems as introduced by Harel and Pnueli \cite{122}. Like interactive systems, reactive systems have an ongoing communication with their environment, but in contrast to interactive systems, reactive systems have to respond to the events generated by their environment at the points of time determined by the environment. For this reason, reactive systems are special kinds of real-time systems so that not only special real-time operating systems are used, but also the worst-case execution/reaction time has to be estimated for these systems.

Since embedded systems were used in safety-critical areas from their beginning, the design flows must take special care on the correctness of the software and hardware developed for such systems. Formal verification is one of the success stories of modern computer science, where in particular model checking makes it now possible to completely verify considerably large systems using sophisticated techniques like symbolic state space representations and various kinds of abstractions. It is therefore mandatory that a design flow covers formal verification and to this end, used languages must have a formal semantics that allows a direct translation to state transition systems as used by modern verification methods.

Like essentially all computer systems today, also the hardware of embedded systems is digital, so that their modeling as discrete state transition systems is adequate. However, the environment of embedded systems is often continuous. Hence, to argue about the effects on the physical environment, one has to consider the entire system which is a hybrid system \cite{7, 125, 126}. In these systems, the state of the environment is determined by differential equations that determine the continuous change of variables describing the environment’s behavior, while the states of the embedded system are discrete. The state transitions of the embedded system may lead to changes of the differential equations, and therefore may influence the environment of the embedded system, while the values of the continuous inputs to the embedded system may influence their state transitions. A holistic consideration
of embedded systems must therefore not necessarily be able to synthesize mixed-signal systems, but it must be able to simulate and verify such systems.

1.2 Models of Concurrent Computation

Traditional programming languages miss several features that are of essential importance for the design of reactive real-time systems. In particular, a notion of time and concurrency is required as well as special data types to distinguish between events and stored values. For this reason, many new languages have been proposed that provide such features. However, only a few of these languages allow both the synthesis of hardware as well as software from the same program, and only a few of them allow a translation to state transition systems as required for most formal verification methods.

In order to classify different kinds of programming languages and modeling styles, models of concurrent computation have been introduced [109, 133, 165, 166]. Being more abstract as programming languages, a model of concurrent computation has to determine what triggers the execution of a concurrent action and how do concurrent parts of a system communicate with each other.

Among many different models of computation, the most important ones are the following ones:

Event-Triggered Languages

The computation of discrete event systems [67] is triggered by the occurrence of an event, which may be the change of a variable’s value, the satisfiability of a desired condition, or the reaching of a desired point of time. Most hardware description languages including VHDL [132], Verilog [130, 179] and SystemC [131] are based on this model of computation, which is best suited for simulation.

Languages that are based on this model of computation typically consist of several sequential processes that are statically defined. The semantics is thereby given by the definition of a simulator that consists of the following phases which define this model of computation:

1. In a first elaboration phase, the active processes are executed until some form of a wait statement is reached. During this elaboration phase, all values for the assignments to variables are determined in the current variable environment, but the values are not yet assigned. Instead, they are maintained in a schedule where the assignments are scheduled at an explicit point of (potentially physical) time.

2. After all processes have been elaborated in a simulation cycle, the assignments scheduled to the current point of time are synchronously executed in an update phase. The possible changes of variable’s values lead to events
that may trigger the further execution of processes at the same point of time.

3. The final step is the event detection phase: If some variable's values have been updated, the next simulation cycle takes place at the same point of (physical time), since new events occurred for the current point of time. Otherwise, the point of time for the next simulation cycle is determined by the next point of time in the schedule where assignments should be executed. The next simulation cycle repeats these steps for the determined point of time until the schedule becomes empty.

The advantage of the above model of computation is that it provides a deterministic form of concurrency: Since the elaboration phase is performed in the same variable environment, the ordering of processes to be elaborated does not matter and potential write conflicts are detected in the update phase.

As any model of computation, there are some intrinsic semantic problems that lead to undesired behaviors or a lack of behavior. In case of discrete event systems, there may be no progress in physical time, since the processes may generate infinitely many events on the current point of time. Moreover, processes may get stuck in infinite loops, which can however be avoided by certain restrictions of the grammars (to demand the execution of wait statements in each loop). Moreover, it can be the case that the schedule requires unbounded memory since processes generate events for later points of time.

Discrete-event based languages lend themselves well for simulation, since the semantics already defines an efficient simulator that executes exactly the necessary changes per execution cycle. However, a synthesis for synchronous hardware circuits is not straightforward unless some restrictions are obeyed. A synthesis of multithreaded software, on the other hand, is straightforward, since the elaboration phase of the processes can be directly implemented by different software threads.

Dataflow Process Networks and CSP

Another model of computation is given by dataflow process networks [138, 139, 141, 164]. Such a network consists of a directed graph whose nodes are associated with simple functions that take some input values from their incoming arcs and produce output values that are put on their outgoing arcs (see Figure 1.2). The firing of the nodes is possible whenever the required data values are available on its incoming arcs. Depending on the state of the process node, and the current input values, it may be necessary that more than one value is required on one input arc, while it may also be possible that no input value is consumed from another arc. The arcs connecting the process nodes are viewed as unbounded FIFO buffers, so that outputs can always be written to these buffers.

For example, consider Berry's Gustave function that requires three input arcs $x_1$, $x_2$, and $x_3$ with boolean values and generates boolean values on a single output arc $y$. The firing rules of this function are defined as follows:
As can be seen by the above table, a node associated with the Gustave function can fire in the three listed cases, and in each case, it produces the same output value on the output arc $y$. For example, this means, the node can fire if $x_1$ and $x_2$ hold the values 0 and 1, respectively, while the content of $x_3$ is irrelevant (it is not read, and therefore, not modified in this case).

As another example, consider a node with the following firing rules:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Depending on the value read from the input buffer $x_1$, either two values must be read from $x_2$ to generate their sum as output value, or a single value has to be read from each of the input buffers $x_2$ and $x_3$ to generate their difference as output value. In this node, the number of data values consumed depends on one of the input values, so that the node is a dynamic dataflow process.

Since the firing of the single process nodes may not be determined in a fixed schedule at compile time, it is usually not controlled by a central control unit. Instead, each process node has to check its input buffers for available data to fire. Therefore, the execution of the entire process network may not be deterministic in that the points of time where output values are generated may not be known in advance. Nevertheless, it is still desirable that the semantics of a dataflow process network should be a function, i.e., deterministic. This semantics is defined as a stream processing function that maps the input
data streams to output data streams. A stream is thereby either a finite or infinite sequence of values of a certain type. Hence, if \( D \) denotes the set of data values, \( D^\omega \) denotes the set of finite and infinite sequences over \( D \), then the semantics of a dataflow process network is a function of type \( (D^\omega)^m \rightarrow (D^\omega)^n \).

The definition of the semantics of a process network is however not that simple, since outputs may be fed back as inputs to the process network. For this reason, the output streams are only incrementally computed which makes it difficult to reason about the determinism of a process network. One therefore considers the prefix ordering of streams where two streams \( \sigma_1, \sigma_2 \) are ordered \( \sigma_1 \preceq \sigma_2 \) iff either \( \sigma_1 = \sigma_2 \) or \( \sigma_1 \) is a finite prefix of \( \sigma_2 \). It is easily seen that the set of streams forms a complete partial order with the prefix ordering.

Based on the prefix ordering of streams, one can easily define monotonic and continuous stream processing functions: a function \( f \) is monotonic iff \( \sigma_1 \preceq \sigma_2 \) implies \( f(\sigma_1) \preceq f(\sigma_2) \), and \( f \) is continuous iff \( f(\operatorname{sup}(M)) = \operatorname{sup}(f(M)) \) holds for every nonempty directed\(^2\) set \( M \). Then, the entire behavior of a dataflow process network can be determined as a least fixpoint of the functions attached to the single process nodes \([138, 139, 241, 243]\) which is known as the Kahn principle \([138, 139]\). Such a fixpoint exists for all continuous functions, so that determinism follows by the continuity of the functions implemented by the single process nodes.

In practice, one often imposes stronger requirements than continuity like blocking-read and lack of emptiness test of input buffers. Testing the emptiness of an input buffer to determine the output of a node is bad, since the node cannot distinguish whether that input stream was finite and has been completely read or whether further input values will arrive, but have not yet arrived. If a read operation to an input buffer is not blocking, it can be used to implement an emptiness test, which is bad for the above reason. Forbidding both emptiness test and unblocking-read guarantees continuous functions, and therefore a deterministic semantics of the dataflow process network.

It is however known that there are continuous functions that do not fulfill these stronger requirements: For example, the above mentioned Gustave function is continuous, but can obviously not be implemented by a blocking-read policy (since it may be the case that one unfortunately tries to read an input stream \( x_i \) that carries no further values). In practice, the blocking-read policy and the lack of emptiness test is however sufficient.

An advantage of Kahn process networks is that their behaviors can be incrementally computed: First, continuous functions are monotonic, and second, due to Bekic’s lemma, it does not matter whether we proceed faster or slower with the consumption of values of the one or the other input stream.

Vuillemin strengthens the notion of continuous function to sequential functions \([164, 189, 258]\). Formally, a function \( f : (D^\omega)^m \rightarrow (D^\omega)^n \) is sequential

\(^1\) We ignore different types in this formulation.

\(^2\) A set \( M \) is directed iff for two elements \( x, y \in M \), there is an element \( z \in M \) such that \( x \preceq z \) and \( y \preceq z \).
if it is continuous and for all input streams $\sigma_1, \ldots, \sigma_m \in D^\omega$, there is an $i \in \{1, \ldots, m\}$ such that for all extended input streams $\sigma'_1, \ldots, \sigma'_m \in D^\omega$ with $\sigma_j \preceq \sigma'_j$ for $j \neq i$ and $\sigma_i \preceq \sigma'_i$, we have $f(\sigma_1, \ldots, \sigma_m) = f(\sigma'_1, \ldots, \sigma'_m)$. This means that the computation of the output streams of $f$ can not proceed until a new value is received from input stream $\sigma_i$. In practice, this means that for a sequential function, one knows at every point of time due to the so-far read input sequences which value has to be read next from which input buffer. This means, for each sequential function, we can start reading its arguments from certain input buffers, and depending on their values, we go on by reading some other channels etc. until all values have been read to compute the current reaction. For example, an if-then-else node is sequential in that it first reads the condition value and depending on its value either the ‘then’ or ‘else’ value. However, Berry’s Gustave function is a continuous function that is not sequential.

Berry defined the set of stable functions [38, 39, 189] that may not be sequential, but that are always continuous.

A particular disadvantage of dataflow process networks are their problematic use of hierarchy (see also [255, 256]): If a subgraph should be collapsed into a single node, its observable behavior appears to be nondeterministic due to different possible schedules of its nodes. Thus, many researchers considered the generalization of dataflow process networks whose process nodes implement nondeterministic behaviors. However, the Keller [143] anomaly demonstrates that fixpoints with set functions can not be used to describe the semantics of these networks, and the Brock-Ackermann anomaly [55] demonstrates that the use of relations instead of functions for the process nodes does also neither lead to a solution. In the meantime, solutions have been found by various researchers that are based on trace-based semantics [23, 147], game-based semantics [99], or more clever functionals [243].

Besides the determinism, another semantic problem is to check whether finite buffers are sufficient to run a dataflow network, and if so, to determine an appropriate schedule for the execution of the process nodes. This question is undecidable for general dataflow networks, but efficient solutions exist for restricted kinds of dataflow networks. The most efficient solution is obtained for synchronous dataflow networks [46, 48, 162?, 163], where the number of input values consumed and produced is statically known (and thus independent of the actual input values) for each process node. For these synchronous dataflow networks, one can determine a topology matrix and based on its kernel, one can determine a periodic schedule that guarantees that finite memory is sufficient to run the network forever.

A generalization of synchronous dataflow networks is given by cyclo-static dataflow networks [49, 85, 96, 97, 199, 262]. In cyclo-static dataflow networks, the consumption and production of data values are periodically constant, since every process node executes a sequence of functions having constant consumption and production rates. Although cyclo-static dataflow net-
works are more general than synchronous ones, it is still possible to construct a schedule with bounded memory at compile time.

While synchronous dataflow networks allow efficient implementations, they are not Turing-complete. It can be shown that the addition of merge and select nodes which leads to Boolean dataflow networks [58–60] leads to Turing-complete networks whose scheduling problems are therefore undecidable. Nevertheless, Park [198] has developed good strategies for increasing the size of finite FIFO buffers in case deadlocks occur at runtime.

### Synchronous Languages

Synchronous languages [28, 34, 118] are becoming more and more attractive for the design and the verification of reactive real-time systems. There are imperative languages like Esterel [41, 45], ECL [154], and Jester [112], data flow languages like Lustre [120] and Signal [36, 107, 156], and graphical languages like some statechart [121] variants [9]. We concentrate in this paper on imperative synchronous languages, in particular on the Esterel family, but note that graphical and imperative synchronous languages can be naturally translated into each other [9].

The basic paradigm of synchronous languages is the perfect synchrony [118], which means that most of the statements are executed in zero time (at least in the idealized programmer’s model). Synchronous computations consist of a possibly infinite sequence of atomic reactions that are indexed by a global logical clock. In each reaction, all inputs are read and all outputs are computed by all components in parallel. In the programmer’s view, the communication and computation of values is done in zero time. Consumption of time must be explicitly programmed with special statements, as e.g. the pause statement in Esterel. Each execution of a pause statement consumes one logical unit of time, and therefore separates different interactions from each other. As the pause statement is the only basic statement that consumes time, it follows that all threads of a synchronous program run in lockstep: they execute the code between pause statements in zero time, and synchronize at the next pause statements. Note that this synchronization is simply due to the semantics of the language.

The control flow of a synchronous program \( P \) can therefore be compiled into a finite state machine \( A_P \) in that we describe how the control flow moves from a set of currently active pause statements to the set of pause statements that are active at the next point of time. Of course, we must also consider the data flow of a program, i.e. how the transition of the control flow manipulates the data values of the program. Therefore, we can model any imperative synchronous program by a finite state control flow that manipulates possibly infinite data types. We call such a model an ‘abstract state machine’ in the following. It is straightforward to convert abstract state machines into a sequential (i.e. single-threaded) imperative programs, as e.g. a C or Java programs.
1 Introduction

[? ? ? ] or to VHDL programs to synthesize a hardware circuit. Therefore, Esterel programs can be both used for hardware or software generation.

The translation of synchronous programs to the corresponding abstract state machines is an essential means for code generation and formal verification. Therefore, a lot of ways have been studied for this translation: [42] distinguishes between a process-algebraic, a finite-state machine, and a hardware circuit semantics. The process-algebraic and the finite state machine semantics are used to enumerate the control states of a program by a depth first traversal so that the abstract state machine is explicitly constructed. Therefore, these translations suffer from the drawback that a program of length $n$ may have $O(n)$ pause statements and therefore $2^{O(n)}$ states. This drawback is circumvented by the hardware circuit semantics in that each program statement is mapped to a corresponding circuit template. This allows a linear time translation of the programs to corresponding hardware circuits.

While synchronous languages like Esterel offer anything that is required for the implementation and verification of reactive real-time systems, there are particular needs for modelling such systems which are not met by synchronous languages. In particular, modern verification methods as e.g. the abstraction from certain data types [? ? ], yield in nondeterministic systems that cannot be directly described by Esterel. Moreover, distributed systems do not obey the synchronous execution of threads so that we must be able to consider asynchronous concurrency as well.

For this reason, we have developed a new 'synchronous' language called Quartz that is very similar to Esterel. In particular, we have added statements for asynchronous parallel execution of threads, and for explicitly implementing nondeterminism. There are also some differences in the semantics of the data values that are used in Quartz and Esterel: We found it important to extend the language with delayed data assignments. These statements work exactly like the immediate versions, but their effect will only take place at the next instant of time. As we will see by our examples, delayed data manipulations allow to conveniently describe many (sequential) algorithms and also hardware circuits. Moreover, we have added in Quartz statements to describe some sort of quantitative time consumption, where 'quantitative' still means that a couple of logical units of time are consumed. In principle, this can also be obtained by sequencing of pause statements, but the use of special analysis tools for analysing such quantitative time bounds requires to describe and translate these constraints explicitly. Finally, we are currently on the way to extend Quartz to handle even analog data so that we will then be even able to deal with hybrid systems [6, 7] as well.

In this article, we present the core of our language Quartz in that we define its syntax and semantics. Concerning the syntax, we do not consider any lexical aspects for the parsing of the language here. The syntax is still likely to change in the future, but the underlying statements together with their semantics yet turns out to be robust.
We have chosen a new way to define the semantics of Quartz that can be directly used both for verification methods like theorem proving and symbolic model checking. The key to our semantics is that we distinguish between the control and data flow of the program, which is a well-known technique for hardware designers. The definition of the control flow is based on the definition of the predicates $\text{enter}_h(S)$, $\text{move}(S)$, and $\text{term}(S)$, that describe entering transitions, internal transitions, and terminating transitions of a statement $S$, respectively. These predicates are then used to define the transition relation of a finite state machine that defines the control flow. The data flow is defined by the guarded commands. These are of the form $(\gamma, C)$, where $C$ is a data manipulating statement that is invoked whenever the condition $\gamma$ holds (Section ??). It is straightforward to label the transitions of the control flow finite state machine by the guarded commands that are enabled by the corresponding transition, so that finally an abstract state machine is obtained that describes the entire semantics of the program (Section ??).

It is interesting to note that all our definitions are simply given by primitive recursion one after the other. This allowed us to easily implement these definitions in the interactive theorem prover HOL [111]. Hence, the Quartz programs may now be used as parts of the higher order logic provided by the HOL systems. Using this embedding in HOL, we can reason at a meta level about the language itself, and also about properties of particular programs. For example, we have proved the correctness of the hardware synthesis as presented in Chapter 6 with the theorem prover, and may thus even use the theorem prover to translate programs to hardware circuits whereas correctness proofs of the translation are generated as a side effect. Hence, we may use our embedding even for a formal synthesis [149] of Quartz programs.

The article is organized as follows: In the next section, we briefly present the syntax of Quartz and briefly discuss their meaning in an informal manner. Then, we formally define the semantics of Quartz, where we distinguish between the control and data flow. We then mention some experimental results that we have obtained by our translations, and define a synthesis method that can be used for code generation (hardware and software). This synthesis method is related to the hardware synthesis of Esterel programs, but is on the one hand more efficient, and circumvents, on the other hand, the problem of schizophrenic synchronizer circuits [42]. We then discuss the issues of causality and reactivity and consider some benchmark examples.

In the appendices, we list the semantics of further macro statements that are used in Esterel as well. We also consider some special constructs, like the use of mutual exclusive regions. We also consider the translation of quantitative time consumption to timed automata. Finally, we list there a process algebraic semantics of Quartz.

- different variants of synchronous languages
- Esterel’s use of $\bot$ or better $\ diamondsuit$: $\bot$ means there is a value, but we do not yet know which one it is, while $\ diamondsuit$ means there is currently no value. Clearly,
this discussion becomes somehow philosophical when we say that is new special value.
• Lustre goes even further and considers streams of data, where is filtered out.
• Signal must even determine clocks to be triggered so that data streams can be computed.

1.3 Model-based Design Methods

1.3.1 High-level Synthesis

1.3.2 Software Synthesis for Embedded Systems

further aspects should be discussed here to motivate synchronous languages:
• finite data-types
• concurrency
• SystemC
• different hardware architectures: GPUs, VLIWs, CPUs, ASIPs, ManyCores
• different code generations: not only HW and SW, but also multi-threaded and pipelined
• from synchrony to asynchrony; relationship to dataflow computing
• very important aspect: formal verification!
• not addressed by most model-based approaches like those based on UML
• well-addressed by SystemC, but no support for code generation is available

1.3.3 Latency-Insensitive Implementations
In this chapter, we define the available data types, program expressions, and specifications of the Quartz language. Before we start the detailed description in the next sections, we have to discuss some general issues in advance.

Since Quartz is a programming language with concurrent actions, it is important to distinguish between atomic and composite data types. The difference between these two kinds of data types is very important for analyzing runtime errors like write conflicts and causality cycles (see Sections 3.2.2, 4.3, 4.6, 8): Atomic actions of the Quartz language work on variables and values of atomic data types, i.e., in a single macro step, more than one atomic action can write to different components of a variable of a composite type. For example, it is possible to concurrently write values to different elements of an array. However, a write conflict is obtained if two assignments were given to a variable of an atomic type. This distinction between atomic and composite data types is the main reason for distinguishing between bitvectors and boolean arrays.

Besides the data type, variable declarations must also determine the information flow of a variable. The information flow classifies variables into input, inout, output, and local variables, which restricts the read and write accesses of a module to these variables: it is not possible to write to input variables, and it is not possible to read output variables. In contrast, inout and local variables can be both read and written. Clearly, local variables have limited scope and are not known outside that scope.

Finally, a storage type must be declared for each variable which is either event or memorized. This distinction becomes important when no action currently determines the value of a variable. In this case, the so-called reaction-to-absence takes place which is different for event and memorized variables: While a memorized variable maintains its previous value, an event variables falls back on a default value (determined by its type).

We consider declarations of variables in Section 3.1.1. In the following, we concentrate on data types that are either atomic or composited, and we have to determine a default value for each data type that is used for initialization of variables and for the reaction-to-absence of event variables.
2.1 Data Types

Quartz is a statically typed programming language, i.e., we can derive for every correctly typed expression a uniquely determined minimal type at compile time that is a property of the expression that does not change during runtime. As in every typed programming language, a data type in Quartz represents a certain set of values. In the following, we define for each of the available types the set of values that are represented by that type to define the semantics of the data type. In Section 2.2.3, we then consider expressions of these data types and define their semantics. The semantics of an expression is a value of the set of values associated with the data type of the expression. We also consider the concrete syntax of types and expressions as well as the rules to derive the minimal types.

2.1.1 Syntax and Semantics of Data Types

There is an inconvenience that we cannot avoid: As can be seen below, the definitions of data types and expressions depend on each other, since some data types are specified with a static expression. A static expression is thereby an expression that can be completely evaluated to a constant value at compile time. For this reason, the reader may assume at the first reading that static expressions are constants, but should keep in mind that arbitrary static expressions can be used instead. Indeed, compilers will usually evaluate in a first step all the static expressions to constants, so that after this first step, static expressions are really constants.

The following enumeration lists all data types, where we already distinguish between atomic and composite data types:

**Definition 2.1 (Data Types).** Expressions in Quartz may have one of the following data types, where \( n \) is a static expression of type \( \text{nat} \) with \( n > 0 \):

**atomic data types:**
- \( \text{bool} \) denotes boolean values \( \text{true} \) and \( \text{false} \)
- \( \text{bv}[n] \) denotes bounded length bitvectors with \( n \) bits
- \( \text{nat}<n> \) denotes bounded unsigned integers
- \( \text{int}<n> \) denotes bounded signed integers
- \( \text{bv} \) denotes unbounded length bitvectors
- \( \text{nat} \) denotes unbounded unsigned integers
- \( \text{int} \) denotes unbounded signed integers

**composite data types:**
- \( \text{array}(\alpha, n) \) denotes arrays of type \( \alpha \) with \( n \) field entries
- \( \alpha \ast \beta \) denotes a tuple type composed of types \( \alpha \) and \( \beta \)

Moreover, the types \( \text{nat}[n] \) and \( \text{int}[n] \) are defined as \( \text{nat}<\exp^2(n)> \) and \( \text{int}<\exp^2(n)> \), respectively.
The above list already makes use of the concrete syntax of types except for arrays that are declared differently (see Chapter 3 for the concrete syntax of variable type declarations). For every data type \( \alpha \), we define its semantics \( \llbracket \alpha \rrbracket_\xi \) as a set of values that may depend on the variable assignment \( \xi \) to evaluate the static expression \( n \) used in the above definition. Variables and, more general, expressions of type \( \alpha \) may then have a value of the set \( \llbracket \alpha \rrbracket_\xi \).

In the definition below, we make use of the set of boolean values \( B = \{ \text{true}, \text{false} \} \), bounded length tuples (sequences) \( \alpha^n \) of length \( n \) as well as unbounded length tuples (sequences) \( \alpha^* := \bigcup_{i=0}^{\infty} \alpha^i \) over a set \( \alpha \). Moreover, we need the sets of natural numbers \( \mathbb{N} = \{ 0, 1, 2, \ldots \} \) (see Appendix B.1) and the set of integers \( \mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \) (see Appendix B.3).

**Definition 2.2 (Formal Semantics of Data Types).** Using a variable assignment \( \xi \) for the static constants, the semantics of the data types of Definition 2.1 is as follows, where \( n := \llbracket n \rrbracket_\xi \) is defined according to Definition 2.5:

- \( \llbracket \text{bool} \rrbracket_\xi := B = \{ \text{true}, \text{false} \} \)
- \( \llbracket \text{bv}[n] \rrbracket_\xi := B^n \)
- \( \llbracket \text{nat}<n> \rrbracket_\xi := \{ 0, \ldots, n-1 \} \)
- \( \llbracket \text{int}<n> \rrbracket_\xi := \{ -n, \ldots, n-1 \} \)
- \( \llbracket \text{bv} \rrbracket_\xi := B^* \)
- \( \llbracket \text{nat} \rrbracket_\xi := \mathbb{N} \)
- \( \llbracket \text{int} \rrbracket_\xi := \mathbb{Z} \)
- \( \llbracket \text{array}(\alpha, n) \rrbracket_\xi \) is the set of functions from \( \{ 0, \ldots, n-1 \} \) to \( \llbracket \alpha \rrbracket_\xi \)
- \( \llbracket \alpha \ast \beta \rrbracket_\xi := \llbracket \alpha \rrbracket_\xi \times \llbracket \beta \rrbracket_\xi \)

According to the above semantics, some types are supertypes or subtypes of other types. For example \( \text{nat}<3> \) is a subtype of \( \text{nat}<5> \). The type system we will describe in the next section will always derive the minimal type of an expression. Thus, the value of the expression will belong at least to the set that is the semantics of its minimal type. However, we can alternatively also use any supertype which is sometimes necessary.

**Definition 2.3 (Type Equivalence and Subtypes).** For types \( \alpha \) and \( \beta \), we define the following binary relations:

- \( \alpha \approx \beta \) holds if \( \llbracket \alpha \rrbracket_\xi = \llbracket \beta \rrbracket_\xi \)
- \( \alpha \preceq \beta \) holds if \( \llbracket \alpha \rrbracket_\xi \subseteq \llbracket \beta \rrbracket_\xi \)

If \( \alpha \approx \beta \) holds, we say that \( \alpha \) and \( \beta \) are equivalent. If \( \alpha \preceq \beta \) holds, we say that \( \alpha \) is a subtype of \( \beta \) and \( \beta \) is a supertype of \( \alpha \).

We will see in the next chapter that the notion of subtypes and supertypes will simplify the type system. For example, it is sufficient to consider addition operators that take either arguments of type \( \text{nat}<n> \) or of type \( \text{int}<n> \). Different range types are simply obtained by switching from a type \( \text{nat}<n> \) to its supertype \( \text{nat}<n+k> \) or from a type \( \text{int}<n> \) to its supertype \( \text{int}<n+k> \). Analogously, the addition of operands of types \( \text{nat}<n> \) and \( \text{int}<m> \) need not be
defined since we can switch from a type \texttt{nat<n>} to its supertype \texttt{int<n>} (but not vice versa). As an alternative to supertypes, we could use more overloaded operators in that we define different versions of the addition operation.

In contrast to previous versions of Quartz, the types \texttt{nat<n>} and \texttt{int<n>} are no longer viewed as bitvectors which offers the possibility to use different encodings of these types for hardware synthesis like signed-digit numbers, radix-2, or 2-complement numbers. The use of ranges instead of the coarser bitwidth yields moreover tighter estimations on the real numbers of bits required for the expressions, so that the compiler can do more overflow checks at compile time.

### 2.1.2 Matching Types to Expected Supertypes

The type rules given in Figures 2.1-2.3 can be used to derive one of the possible types of an expression. If we exclude the type rules (I.1)-(I.8), the minimal type would be computed. However, in many expressions this would not allow us to derive any type at all even though the expression is correctly typed. This is due to the fact that we did not list all possible cases of argument types of all operators, and instead only listed some maximal argument types. For this reason, the rules (I.1)-(I.8) are necessary to derive even the minimal type.

For example, consider the expression \texttt{4-2u}. According to the type rules of literals, we obtain \texttt{4:int<5>} and \texttt{2u:nat<3>}. The type rules for subtraction (II.17) and (II.18) do however not allow arguments of that type. For this reason, we have to apply rule (I.7) to lift the type of \texttt{2u} to the supertype \texttt{2u:int<3>}. Let us call types that only differ in their ranges homogeneous i.e., \texttt{nat<3>}, \texttt{nat<6>} and \texttt{nat} as well as \texttt{int<3>}, \texttt{int<6>} and \texttt{int}.

In cases where two or three arguments of heterogeneous types are given, but where only type rules of homogeneous types are available, we can use the following table to compute the maximum type where all argument types can be lifted to:

<table>
<thead>
<tr>
<th></th>
<th>bv[n]</th>
<th>nat&lt;n&gt;</th>
<th>int&lt;n&gt;</th>
<th>bv nat int</th>
</tr>
</thead>
<tbody>
<tr>
<td>bv[m]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>nat&lt;n&gt;</td>
<td>–</td>
<td>nat&lt;max(m,n)&gt;</td>
<td>int&lt;max(m,n)&gt;</td>
<td>–</td>
</tr>
<tr>
<td>int&lt;n&gt;</td>
<td>–</td>
<td>int&lt;max(m,n)&gt;</td>
<td>int&lt;max(m,n)&gt;</td>
<td>–</td>
</tr>
<tr>
<td>bv</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>nat</td>
<td>–</td>
<td>nat</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>int</td>
<td>–</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

Of course, the table can be extended also to composite types, so that we can also compute maximum types for tuple and array types.

Hence, the use of an expression in a context also often leads to an expected type that must, in general, be a supertype of the already derived type of the expression. The compiler can then extend the type to the supertype so that the generated typed expression obeys the type rules of Quartz.
However, there are some exceptions of this rule, which should from a precise theoretical point of view not be allowed. Nevertheless, it turns out to be too inconvenient in practice if we would not break the rules for an expression $\tau$ in the following cases:

- assignment of $\tau$ to a variable $x = \tau$
- use of $\tau$ as index expression of an array expression $x[\tau]$
- use of $\tau$ as argument of a module or function call $\text{name}(\ldots, \tau, \ldots)$

From a theoretical point of view, we could always use modulo or other operations to map $\tau$ to the required range. However, in practice this turned out to be too inconvenient, so that we decided to be sloppy in the above cases. The classic example is the incrementation of a variable $\text{next}(i) = i+1$ that can never be correctly typed with bounded numeric types, since the expression $i+1$ has type $\text{nat}<n+1>$ if $i$ has type $\text{nat}<n>$. Compilers should therefore offer options to automatically generate assertions to check in the above cases whether the ranges are sufficient, which can be done in simple cases by static type checking, but requires in general formal verification methods. We describe in the sections on hardware and software synthesis how our current Quartz compiler handles these cases in code generation.

### 2.1.3 Primitive Recursive Data Types

Primitive recursive data types such as lists or trees are not yet available in Quartz. It is planned to add primitive recursive data types in future versions. To this end, we will follow the styles as implemented in higher order theorem provers which fulfills the highest standards to obtain high quality code.

### 2.2 Expressions

In this section, we define the set of expressions that can be used in programs. Every expression has a uniquely determined minimal type, and the semantics of an expression is a value of the set that is the semantics of the type of the expression. In the following, we often say that the value of the expression is consistent with its type to express that the value is a member of the set that is the semantics of the corresponding type.

In this section, we will first consider the syntax of expressions in that we present the list of operators together with lexical rules like precedences and associativities. Then we will describe in detail how the minimal type of an expression is obtained. As in every typed programming language, there are expressions that are not type-consistent, i.e., expressions where no type can be inferred. The purpose of type-checking is to make sure that the operators are applied correctly and to check to some extent that the bitwidths are sufficient.

1. This is not seen after textual expansion of the calls!
The semantics of an expression is a value consistent with the type of the expression. If the expression does not contain variables, it is a static expression. Static expression can be fully evaluated at compile time to values of a certain minimal type. It has to be noted that the minimal type of a static expression can only be determined after its evaluation to a constant value. The definition of the semantics describes in this case how the static evaluation is to be performed to obtain this value and its minimal type.

Expressions with variables depend on the values of these variables. Hence, the semantics of such expressions depends on an assignment \( \xi \) of these variables to values (which is the content of the memory in technical realizations). We will describe both the static and dynamic evaluation in a common setting.

In the following, we consider for different syntactic objects the aspects of concrete syntax, type checking rules, and the semantics. To specify the type inference, we write \( \tau : \alpha \) to express that the type \( \alpha \) has been derived for expression \( \tau \). The semantics of an expression \( \tau \) is written as \( [\tau]_{\xi} \).

### 2.2.1 Variables

#### Concrete Syntax

Variables are given as strings where digits ‘0’–‘9’, lower case letters \(^2\) ‘a’–‘z’, upper case letters ‘A’–‘Z’, and the underscore symbol ‘_’ are allowed. However, not all nonempty sequences of these characters are allowed, there are the following restrictions:

- The identifier must start with a lower or upper case letter.
- The underscore character must not follow another underscore character.

The restriction to only allow single occurrences of underscore characters is used to offer compilers a simple way to produce new identifiers when necessary in that the compiler generated identifiers use a subsequence of two underscore characters.

#### Type Checking

The type of a variable must be specified in its declaration (see Chapter 3). Therefore, there is only one typing rule that simply refers to the given declaration:

\[
\text{variable } x \text{ declared with type } \alpha
\]

\[
\frac{}{x : \alpha}
\]

---

\(^2\) A letter is one of the following symbols: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, and digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Hence, there are no Latin1 symbols and no Unicode symbols.
2.2 Expressions

Semantics

The semantics \([x]_\xi\) of a variable \(x\) depends on the variable assignment, i.e., we define \([x]_\xi := \xi(x)\). We assume that the variable assignment \(\xi\) respects the type of the variable. This means that if \(x\) has type \(\alpha\), then \(\xi(x) \in \theta[\alpha]_\xi\).

2.2.2 Literals

Concrete Syntax

Literals are strings that describe constant values. There is no need to specify the type of a literal, since the literals are chosen so that the type can be implicitly derived. For the different data types, the following literals are allowed:

- **bool**: The two literals true and false are the only constants of type bool.
- **bv\([n]\)**: Constants of bitvector type bv\([n]\) can be given in hexadecimal, octal, or binary representation. To distinguish these from constants int\(<n>\), one of the characters x, o, or b has to be added as a suffix to indicate a constant given in hexadecimal, octal, and binary representation, respectively. Examples for bitvector constants are therefore 101010b, 52o, and 2AFx.
- **int\(<n>\)**: Constants of bitvector type int\(<n>\) are given as decimal numbers with a potential leading negative sign '-', i.e., nonempty strings of digits with a potential leading negative sign '-'.
- **nat\(<n>\)**: Constants of bitvector type nat\(<n>\) are given as decimal numbers followed with a character u, i.e., as a nonempty string of digits. The suffix "u" is used to distinguish between constants of types nat\(<n>\) and int\(<n>\). This distinction is necessary to correctly evaluate expressions like 2-3<0: According to the syntax, these are interpreted as constants of type int\(<n>\). Hence, the static evaluation of the compiler first evaluates 2-3 to -1, which is less than 0. Interpreting the constants with type nat\(<n>\) would lead to a different result: the compiler would evaluate 2u-3u to 0u, which is not less than 0u.

It is not possible to write numeric constants as binary, octal or hexadecimal numbers. However, since one can specify such bitvectors as constants of type bv\([n]\), one can then simply convert these constants to constants of type nat\(<n>\) or int\(<n>\) using the functions bv2nat and bv2int as described below. In the same way, one can use the inverse functions nat2bv and int2bv for converting numeric constants to bitvectors.

There are no constants for the unbounded types bv, nat, and int. However, note that the unbounded length bitvector types bv, nat, and int are supersets of the corresponding bounded length types, and therefore the constants of the bounded types also belong to the corresponding unbounded type.
Type Checking

The types of the boolean constants true and false is clear. The types of bitvector constants can be directly derived from the syntax of the constant and the number of the used bits. This leads to the following typing rules, where we use the notation \( \langle \sigma \rangle_B \) to evaluate radix-\( B \) numbers as defined in Definition ?? on page ??:

\[
\begin{align*}
\frac{}{\text{true: bool}} & \quad \frac{}{\text{false: bool}} \\
\frac{\sigma \in \{0,1\}^n}{\sigma : \text{bv}[n]} & \quad \frac{\sigma \in \{0,\ldots,7\}^m}{\sigma : \text{bv}[3\times n]} & \quad \frac{\sigma \in \{0,\ldots,9, A, \ldots, F\}^m}{\sigma : \text{bv}[4\times n]} \\
\frac{\sigma \in \{0,\ldots,9\}^m}{\sigma : \text{int}<\langle \sigma \rangle_{10} + 1>} & \quad \frac{\sigma \in \{\ldots\}'}{\sigma : \text{int}<\langle \sigma \rangle_{10}>} \\
\frac{}{\sigma \in \{0,\ldots,9\}^m u} & \quad \frac{}{\sigma : \text{nat}<\langle \sigma \rangle_{10} + 1>} 
\end{align*}
\]

Semantics

The semantics of the boolean constants true and false is clear: We define \( \|\text{true}\|_\xi := \text{true} \) and \( \|\text{false}\|_\xi := \text{false} \).

The semantics of a bitvector constant \( c \) given as a binary string is analogously obtained in that 1 is mapped to true and 0 is mapped to false to obtain the semantics \( \|c\|_\xi \in \mathbb{B}^n \). The semantics of bitvector constants given as octal strings is obtained by first converting these strings to an equivalent binary string which is done digitwise according to the following table:

<table>
<thead>
<tr>
<th>Octal Digit</th>
<th>Binary Sequence</th>
<th>Octal Digit</th>
<th>Binary Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

Analogously, the semantics of bitvector constants given as hexadecimal strings is obtained by first converting these strings to an equivalent binary string which is done digitwise according to the following table:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Bin</th>
<th>Hex</th>
<th>Bin</th>
<th>Hex</th>
<th>Bin</th>
<th>Hex</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>0100</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>0101</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>A</td>
<td>0110</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>B</td>
<td>0111</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

The semantics of a constant of types \( \text{nat}<n> \) and \( \text{int}<n> \) is clear: the constants are given as decimal numbers.
2.2.3 Operators

As usual for many programming languages, operators can be overloaded, i.e., we use the same symbols for different, but related functions whenever these functions can be distinguished by the types of their arguments. For example, we use the same symbols for signed and unsigned arithmetic operations although the underlying functions significantly differ (see Appendix B).

Table 2.1 lists all available operators together with their precedence (which is important to save delimiters for parsing) and associativity (for binary operators only). Precedence 1 is the highest and 12 the lowest, so that \( a \land b \lor c \) is a disjunction whose left hand argument is a conjunction. One has to use delimiters like in \( a \land (b \lor c) \) to construct a conjunction whose right hand argument is a disjunction.

Associativities of binary operators is a further means to avoid ambiguity in expressions. As \( \land \) and \( \lor \) is left-associative, the expression \( a \land b \land c \) is parsed as \( (a \land b) \land c \) if desired. We followed the definition of the C programming language in that we used the same syntax, precedences, and associativities whenever possible. There are however minor differences: In contrast to the C programming language, comparison operators like \(<\) are not associative in Quartz, since they can not be nested unlike in C. However, equality \(==\) and inequality can be nested. For example, \( a == b == c \) will be read as \( (a == b) == c \) which implies that \( c \) must have type \( bool \), and that \( a \) and \( b \) have the same type.

For boolean operators, one there is the alternative syntax \( !, \&\&, |, \lor, ->, \land\lor, <-> \) instead of \( not, and, or, imp, xor, and equ \). Note that \( == \) and \( != \) are equivalent to \(<->\) and \( xor \), respectively. Nevertheless, we recommend the use of \(<->\) and \( xor \), since these operators can also be bitwisely applied to bitvectors, which is not the case with \( == \) and \( != \) (these return boolean values in this case).

Typing Rules

Table 2.1 presents the syntax of operators that includes the precedences and associativity rules. Clearly, not every expression that can be formed by these syntax rules is a legal expression, since expressions have to be additionally correctly typed. The type system used in Quartz can be summarized in a short hand notation and is shown in Figures 2.1-2.3, where \( \tau : \alpha \) means that the type \( \alpha \) can be derived for expression \( \tau \).

Figure 2.1 presents implicit and explicit type conversion rules. Note that the type of an expression is not uniquely determined, since the rules of Figure 2.1 allow us to switch from one type to another one. However, there is always a minimal type with respect to the partial order relation of Definition 2.3, and this type is uniquely determined. We therefore distinguish between a type of an expression or its minimal type. Rules (I.1)-(I.10) present the implicit type conversion rules that do not require operators and are used
### Table 2.1. Operators with their Precedence and Associativity

<table>
<thead>
<tr>
<th>prec.</th>
<th>operator</th>
<th>associativity</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>τ.m</td>
<td>tuple constructor</td>
<td>tuple access</td>
</tr>
<tr>
<td></td>
<td>a[τ]</td>
<td>array access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ{m}</td>
<td>bitvector bit access</td>
<td>bitvector segment</td>
</tr>
<tr>
<td></td>
<td>τ{m:n}</td>
<td>bitvector segment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ{:n}</td>
<td>bitvector segment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ{m:}</td>
<td>bitvector segment</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>not φ , !φ</td>
<td>bitwise/boolean negation</td>
<td>unary minus</td>
</tr>
<tr>
<td></td>
<td>-τ</td>
<td>absolute value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>abs(τ)</td>
<td>convert int to bv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int2bv(τ)</td>
<td>convert nat to bv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bv2int(τ)</td>
<td>convert bv to int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bv2nat(τ)</td>
<td>convert bv to nat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeOf(τ)</td>
<td>bitwidth of τ in bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ϕ::n}</td>
<td>replicate ϕ n times</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reverse(τ)</td>
<td>reverse bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fromArray(τ)</td>
<td>convert boolean array to bitvector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sat&lt;n&gt;(τ)</td>
<td>saturate τ to nat&lt;n&gt; or int&lt;n&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exp2(τ)</td>
<td>compute $2^\tau$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>log2(τ)</td>
<td>compute $\lceil \log_2(\tau) \rceil$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>τ @ π</td>
<td>left to right bit/bitvector concatenation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ * π</td>
<td>left to right multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ / π</td>
<td>left to right division</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ % π</td>
<td>left to right remainder</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>τ + π</td>
<td>left to right addition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ - π</td>
<td>left to right subtraction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>τ &lt; π</td>
<td>nonassociative less</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ &lt;= π</td>
<td>nonassociative less-or-equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ &gt; π</td>
<td>nonassociative greater</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ &gt;= π</td>
<td>nonassociative greater-or-equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ == π</td>
<td>left to right equality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>τ != π</td>
<td>left to right inequality</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>φ and ψ , φ &amp; ψ</td>
<td>left to right (bitwise) conjunction</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>φ xor ψ</td>
<td>left to right (bitwise) exclusive disjunction</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>φ or ψ , φ</td>
<td>left to right (bitwise) disjunction</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>φ imp ψ , φ -&gt; ψ</td>
<td>left to right (bitwise) implication</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>φ equ ψ , φ -&gt; ψ</td>
<td>left to right (bitwise) equivalence</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(φ?τ:π)</td>
<td>conditional operator</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>forall(α i=m .. n) φ</td>
<td>generic conjunction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exists(α i=m .. n) φ</td>
<td>generic disjunction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum(α i=m .. n) τ</td>
<td>generic addition</td>
<td></td>
</tr>
</tbody>
</table>

by the type checker itself whenever necessary. Without these rules, the minimal type of an expression is derived, while the use of these rules switches to a supertype. In particular, rule (I.1) is the embedding of nat<n> in the supertype nat<n+m> and rule (I.2) is the analogous rule for the types int<n> and nat<n+m>. In binary representations, these rules correspond with digit
2.2 Expressions

### Implicit Type Conversion

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I.1)</td>
<td>$\tau : \text{nat}^n \quad \tau : \text{nat}^{n+m}$</td>
</tr>
<tr>
<td>(I.2)</td>
<td>$\tau : \text{int}^n \quad \tau : \text{int}^{n+m}$</td>
</tr>
<tr>
<td>(I.3)</td>
<td>$\tau : \text{bool} \quad \tau : \text{bv}^1$</td>
</tr>
<tr>
<td>(I.4)</td>
<td>$\tau : \text{bv}^n \quad \tau : \text{bv}$</td>
</tr>
<tr>
<td>(I.5)</td>
<td>$\tau : \text{int}^n \quad \tau : \text{int}$</td>
</tr>
<tr>
<td>(I.6)</td>
<td>$\tau : \text{nat}^n \quad \tau : \text{nat}$</td>
</tr>
<tr>
<td>(I.7)</td>
<td>$\tau : \text{nat}^n \quad \tau : \text{int}^n$</td>
</tr>
<tr>
<td>(I.8)</td>
<td>$\tau : \text{nat} \quad \tau : \text{int}$</td>
</tr>
<tr>
<td>(I.9)</td>
<td>$\tau : \text{nat}^n \quad \tau : \text{nat}^{2^n}$</td>
</tr>
<tr>
<td>(I.10)</td>
<td>$\tau : \text{int}^n \quad \tau : \text{int}^{2^n-1}$</td>
</tr>
</tbody>
</table>

### Explicit Type Conversion

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I.11)</td>
<td>$\tau : \text{bv}^n \quad \text{bv2nat}(\tau) : \text{nat}^{\exp_2(n)}$</td>
</tr>
<tr>
<td>(I.12)</td>
<td>$\tau : \text{bv} \quad \text{bv2nat}(\tau) : \text{nat}$</td>
</tr>
<tr>
<td>(I.13)</td>
<td>$\tau : \text{bv}^n \quad \text{bv2int}(\tau) : \text{int}^{\exp_2(n-1)}$</td>
</tr>
<tr>
<td>(I.14)</td>
<td>$\tau : \text{bv} \quad \text{bv2int}(\tau) : \text{int}$</td>
</tr>
<tr>
<td>(I.15)</td>
<td>$\tau : \text{nat}^n \quad \text{nat2bv}(\tau) : \text{bv}^{\text{natsize}(n-1)}$</td>
</tr>
<tr>
<td>(I.16)</td>
<td>$\tau : \text{nat} \quad \text{nat2bv}(\tau) : \text{bv}$</td>
</tr>
<tr>
<td>(I.17)</td>
<td>$\tau : \text{int}^n \quad \text{int2bv}(\tau) : \text{bv}^{\text{intsize}(-n)}$</td>
</tr>
<tr>
<td>(I.18)</td>
<td>$\tau : \text{int} \quad \text{int2bv}(\tau) : \text{bv}$</td>
</tr>
<tr>
<td>(I.19)</td>
<td>$\tau : \text{int}^n \quad \text{abs}(\tau) : \text{nat}^{n+1}$</td>
</tr>
<tr>
<td>(I.20)</td>
<td>$\tau : \text{int} \quad \text{abs}(\tau) : \text{nat}$</td>
</tr>
</tbody>
</table>

---

**Fig. 2.1.** Type System of Quartz (Part I)

extensions as explained in Lemma B.25. Rules (I.3) states that `bool` is also viewed as `bv^1`, and rules (I.4)-(I.6) state that the bounded types `bv^n`, `nat^n`, and `int^n` are included in the corresponding unbounded types `bv`, `nat`, and `int`. Rules (I.9) and (I.10) introduce the shorthand notation `nat^n`, and `int^n` to specify bounded numeric types via their binary bitwidths.

Rules (I.11)-(I.20) describe explicit type conversion rules, i.e., operators that are used to change a type of an expression with an according conversion function. Rules (I.11) and (I.12) convert bitvectors to natural numbers, where the bitvector is viewed as a radix-2 number. Similarly, rules (I.13) and (I.14) convert bitvectors to integers, where the bitvector is viewed as a 2-complement number. Note that $n$ bit radix-2 numbers encode the $2^n$ numbers $-2^{n-1}, \ldots, 2^{n-1} - 1$ which is the semantics of the type `int^{\exp_2(n-1)}`. The
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(II.1)</td>
<td>$\tau_{n-1} : \alpha_{n-1} \ldots \tau_0 : \alpha_0$</td>
</tr>
<tr>
<td>(II.2)</td>
<td>$\tau : \alpha_{n-1} \ldots \alpha_0$ $\vdash \tau \pi : \alpha[n]$</td>
</tr>
<tr>
<td>(II.3)</td>
<td>$\tau : \text{array}(\alpha, n)$ $\vdash \tau[\pi] : \alpha$</td>
</tr>
<tr>
<td>(II.4)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[m]$</td>
</tr>
<tr>
<td>(II.5)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$</td>
</tr>
<tr>
<td>(II.6)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi_1] : \text{int}&lt;n&gt;$ $\vdash \tau[\pi_2] : \text{int}&lt;n&gt;$ $\vdash \tau[\pi_1 : \pi_2] : \text{bv}[p_1 + p_2 + 1]$</td>
</tr>
<tr>
<td>(II.7)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau[\pi : \pi] : \text{bv}[p_1 + p_2]$</td>
</tr>
<tr>
<td>(II.9)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.10)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.11)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.12)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.13)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.14)</td>
<td>$\tau : \text{bv}[n]$ $\vdash \tau[\pi] : \text{bv}[n]$</td>
</tr>
<tr>
<td>(II.15)</td>
<td>$\tau : \text{nat}&lt;m&gt;$ $\vdash \tau[\pi] : \text{nat}&lt;n&gt;$ $\vdash \tau + \pi : \text{nat}&lt;m+n-1&gt;$</td>
</tr>
<tr>
<td>(II.16)</td>
<td>$\tau : \text{int}&lt;m&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau + \pi : \text{int}&lt;m+n&gt;$</td>
</tr>
<tr>
<td>(II.17)</td>
<td>$\tau : \text{nat}&lt;m&gt;$ $\vdash \tau[\pi] : \text{nat}&lt;n&gt;$ $\vdash \tau - \pi : \text{nat}&lt;m&gt;$</td>
</tr>
<tr>
<td>(II.18)</td>
<td>$\tau : \text{int}&lt;m&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau - \pi : \text{int}&lt;m+n&gt;$</td>
</tr>
<tr>
<td>(II.19)</td>
<td>$\tau : \text{nat}&lt;m&gt;$ $\vdash \tau[\pi] : \text{nat}&lt;n&gt;$ $\vdash \tau * \pi : \text{nat}&lt;m-1*n+1&gt;$</td>
</tr>
<tr>
<td>(II.20)</td>
<td>$\tau : \text{int}&lt;m&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau * \pi : \text{int}&lt;m*n+1&gt;$</td>
</tr>
<tr>
<td>(II.21)</td>
<td>$\tau : \text{nat}&lt;m&gt;$ $\vdash \tau[\pi] : \text{nat}&lt;n&gt;$ $\vdash \tau / \pi : \text{nat}&lt;m&gt;$</td>
</tr>
<tr>
<td>(II.22)</td>
<td>$\tau : \text{int}&lt;m&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau / \pi : \text{int}&lt;m+1&gt;$</td>
</tr>
<tr>
<td>(II.23)</td>
<td>$\tau : \text{nat}&lt;m&gt;$ $\vdash \tau[\pi] : \text{nat}&lt;n&gt;$ $\vdash \tau \div \pi : \text{nat}&lt;n-1&gt;$</td>
</tr>
<tr>
<td>(II.24)</td>
<td>$\tau : \text{int}&lt;m&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau \div \pi : \text{int}&lt;n-1&gt;$</td>
</tr>
<tr>
<td>(II.25)</td>
<td>$\tau : \text{nat}&lt;n&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau \div \pi : \text{int}&lt;n&gt;$</td>
</tr>
<tr>
<td>(II.26)</td>
<td>$\tau : \text{int}&lt;n&gt;$ $\vdash \tau[\pi] : \text{int}&lt;n&gt;$ $\vdash \tau \div \pi : \text{int}&lt;n+1&gt;$</td>
</tr>
<tr>
<td>(II.27)</td>
<td>$\tau : \text{int}$ $\vdash \tau &lt; \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.28)</td>
<td>$\tau : \text{int}$ $\vdash \tau &gt; \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.29)</td>
<td>$\tau : \text{int}$ $\vdash \tau &lt;= \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.30)</td>
<td>$\tau : \text{int}$ $\vdash \tau &gt;= \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.31)</td>
<td>$\tau : \text{int}$ $\vdash \tau == \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.32)</td>
<td>$\tau : \text{int}$ $\vdash \tau != \pi : \text{int}$</td>
</tr>
<tr>
<td>(II.33)</td>
<td>$\tau : \text{int}$ $\vdash \tau [\pi] : \text{int}$</td>
</tr>
</tbody>
</table>

**Fig. 2.2.** Type System of Quartz (Part II)
inverse conversions are typed in rules (I.15)-(I.18), where \( \text{natsize}(n) \) and \( \text{intsize}(n) \) are defined below. Finally, rules (I.19) and (I.20) show how the absolute value function is used to convert signed to unsigned integers.

**Definition 2.4 (Minimal Bitwidths).** For every \( n \in \mathbb{N} \) and every \( m \in \mathbb{Z} \), we define the number \( \text{natsize}(n) \in \mathbb{N} \) and \( \text{intsize}(m) \in \mathbb{N} \) as follows:

- \( \text{natsize}(n) := \begin{cases} \lceil \log_2(n + 1) \rceil & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases} \)
- \( \text{intsize}(n) := \begin{cases} 1 + \lceil \log_2(n + 1) \rceil & \text{for } n > 0 \\ 1 & \text{for } n = 0 \\ 1 + \lceil \log_2(-n) \rceil & \text{for } n < 0 \end{cases} \)

As shown in Appendix B, \( \text{natsize}(n) \) is the minimal number of bits required to represent the number \( n \in \mathbb{N} \) in binary format, and \( \text{intsize}(n) \) is the minimal number of bits required to represent the number \( n \in \mathbb{Z} \) in 2-complement.

Rule (II.1) is the type rule of the tuple constructor that generates, e.g., from two expressions \( \tau \) and \( \pi \) the pair \((\tau, \pi)\). Rule (II.2) is the type rule of the tuple selector operator that selects from a tuple like \((\tau, \pi)\) a component like \((\tau, \pi)_1 = \pi\).

Rule (II.3) refers to array access, where the expression \( a[\tau] \) is allowed to also have array type. Note however that except for array access there is no operator that would accept an expression of array type. Moreover, also the assignment statement is only allowed for atomic types. Note that the type \( \text{int}<n> \) is allowed for the index expression as will be discussed in the next section.

Rules (II.4)-(II-8) are type rules for bitvector expressions. Rule (II.4) describes the construction of bitvectors by the concatenation operator. Rule (II.5) defines the access to a single bit of a bitvector expression. Rules (II.6)-(II.8) define the types of different versions of the slicing operator. Note that
the index expressions $\pi$, $\pi_1$, and $\pi_2$ are static expressions which is a major distinction to array accesses. Analogous to array accesses, we also allow the type $\text{int}<\alpha>$ for the index expressions.

Rules (II.9)-(II.14) describe the type rules for the (bitwise) boolean operators. Note that due to rule (I.3) these operators can also be applied to arguments of type $\text{bool}$. Rules (II.15)-(II.26) are the type rules of the arithmetic operators that should be clear. The ranges of the used types are obtained by considering the minimal and maximal values or -1 (as in case of (II.11)). The rules (II.9)-(II.14) do also apply for unbounded types. The rules for arithmetic relations (II.27)-(II.34) are only stated for type $\text{int}$. However, as $\text{nat}<m>$, $\text{nat}$, and $\text{int}<m>$ are subtypes of $\text{int}$, we can use these rules also for these subtypes in any combination. Note that the equality of bitvectors accepts argument types of different bitwidths (the result is false for different bitwidth).

Rules (III.1) and (III.2) describe possible uses of the $\text{sizeOf}$ operator that can compute the minimal numbers of bits required to implement expressions of types $\text{nat}<n>$ and $\text{int}<m>$ as radix-2 and 2-complement numbers, respectively. (III.3) is the typing of the $\text{reverse}$ operator. Rule (III.4) describes the bit-iterator that generates $n$ copies of a boolean expression in a bitvector. Rule (III.5) explains how the $\text{fromArray}$ operator converts a boolean array to a bitvector. Note that the inverse operation can be easily obtained by a generic sequence of assignments as explained in the section on statements. Rules (III.6) and (III.7) are the type rules for the $\text{sat}$ operator that implements saturated arithmetics as explained in the next section. The remaining rules should be clear.

Note that we do not list type rules for generic conjunction, generic disjunction, and generic sums, since these are viewed as abbreviations of expressions whose type rules have already been given.

Semantics

Having explained the syntax and the type inference rules of expressions, it remains to formally define the meaning of expressions. In general, we define the semantics $\llbracket \tau \rrbracket_\xi$ of an expression $\tau$ with respect to a variable assignment $\xi$ and assume that the variable assignment $\xi$ is type consistent, i.e., that for every variable $x$, the value $\xi(x)$ belongs to the semantics of the type of $x$. This holds also for the semantics $\llbracket \tau \rrbracket_\xi$ of an expression $\tau$, i.e., also $\llbracket \tau \rrbracket_\xi$ is an element of the set of values that is associated with the type of $\tau$.

In general, the semantics $\llbracket \tau \rrbracket_\xi$ of an expression $\tau$ depends on $\xi$, so that whenever the variable assignment changes, also the semantics $\llbracket \tau \rrbracket_\xi$ may be changed. Therefore, the semantics of an expression is rather a function that depends on the expression and on a variable assignment. This is in particular convenient to describe the change of the semantics of an expression during the execution of the program where the values of the variables are changed.
Certain subexpressions must be static expressions, i.e., they must not depend on the current variable assignments. Therefore the values of static expressions can be evaluated at compile time. In case that \( \tau \) must be a static expression, we write \( \llbracket \tau \rrbracket \), thus omitting the dependence on the variable assignment \( \xi \), to enforce the static evaluation of \( \tau \). Static expressions may therefore be viewed as constants, although the may consist of variables that are not relevant for their value. For example, an extreme example is the number of bits \( \text{sizeOf}(x) \) of a variable \( x \) which is a static value although it depends on the variable \( x \).

Using these remarks on the notation, the semantics of expressions is recursively defined below. It is essentially the dynamic (or static) evaluation of expressions and therefore forms a core of the semantics of Quartz.

**Definition 2.5 (Semantics of Expressions).** For a given variable assignment \( \xi \), the semantics of an expression \( \tau \) is a value \( \llbracket \tau \rrbracket_\xi \) that belongs to the set that is the semantics of the type of \( \tau \). This value \( \llbracket \tau \rrbracket_\xi \) is recursively defined as follows:

- for variables and constants, we define
  - \( \llbracket x \rrbracket_\xi := \xi(x) \) for every variable \( x \)
  - \( \llbracket \xi \rrbracket_\xi \) is defined as explained in Section 2.2.2
- for tuple constructor and selector, we define
  - \( \llbracket (\tau_n-1, \ldots, \tau_0) \rrbracket_\xi := (\llbracket \tau_{n-1} \rrbracket_\xi, \ldots, \llbracket \tau_0 \rrbracket_\xi) \)
  - \( \llbracket \tau \times \rrbracket_\xi := \vartheta_p \) where \( \vartheta := (\vartheta_{n-1}, \ldots, \vartheta_0) := \llbracket \tau \rrbracket_\xi \) and \( p := \llbracket \tau \rrbracket \) with \( 0 \leq p < n \)
- for array access, we define
  - \( \llbracket x[\tau] \rrbracket_\xi := [x]_\xi \left( \llbracket \tau \rrbracket_\xi \mod n \right) \right) \), where \( n \) is the size of the array \( x \)
- for \( \vartheta := (\vartheta_{n-1}, \ldots, \vartheta_0) := \llbracket \varphi \rrbracket_\xi \), \( \nu := (\nu_{n-1}, \ldots, \nu_0) := \llbracket \psi \rrbracket_\xi \), \( p_1 := (\llbracket \pi_1 \rrbracket \mod n) \), and \( p_2 := (\llbracket \pi_2 \rrbracket \mod n) \), we define the semantics of bitvector operators as follows:
  - \( \llbracket \varphi \land \psi \rrbracket_\xi := (\vartheta_{n-1}, \ldots, \vartheta_0, \nu_{n-1}, \ldots, \nu_0) \)
  - \( \llbracket \varphi \lor \psi \rrbracket_\xi := \vartheta_p \)
  - \( \llbracket \varphi \land \pi_1 \rrbracket_\xi := (\vartheta_{p_1}, \ldots, \vartheta_0) \)
  - \( \llbracket \varphi \land \pi_2 \rrbracket_\xi := (\vartheta_{p_1}, \ldots, \vartheta_0) \)
  - \( \llbracket \varphi \land \pi_2 \rrbracket_\xi := (\vartheta_{n-1}, \ldots, \vartheta_0) \)
  - \( \llbracket \text{fromArray}(x) \rrbracket_\xi := ([x]_\xi (n-1), \ldots, [x]_\xi (0)), \) where \( n := \llbracket n \rrbracket_\xi \) for \( x : \text{array}(\text{bool}, n) \)
  - \( \llbracket \text{reverse}(\varphi) \rrbracket_\xi := (\vartheta_0, \ldots, \vartheta_{n-1}) \)
  - \( \llbracket \{ \tau : n \} \rrbracket_\xi := (\llbracket \tau \rrbracket_\xi, \ldots, \llbracket \tau \rrbracket_\xi) \) (a tuple with \( \llbracket n \rrbracket \) entries)
  - \( \llbracket \text{sizeOf}(\varphi) \rrbracket_\xi := n \)
- for (bitwise) boolean operators, we define with \( \vartheta := (\vartheta_{n-1}, \ldots, \vartheta_0) := \llbracket \varphi \rrbracket_\xi \) and \( \nu := (\nu_{n-1}, \ldots, \nu_0) := \llbracket \psi \rrbracket_\xi \) the following:
  - \( \llbracket \lnot \varphi \rrbracket_\xi := (\vartheta_{n-1}, \ldots, \vartheta_0) \)
  - \( \llbracket \varphi \land \psi \rrbracket_\xi := (\vartheta_{n-1} \land \vartheta_0, \ldots, \vartheta_{n-1} \land \vartheta_0) \)
  - \( \llbracket \varphi \lor \psi \rrbracket_\xi := (\vartheta_{n-1} \lor \vartheta_0, \ldots, \vartheta_{n-1} \lor \vartheta_0) \)
  - \( \llbracket \varphi \Rightarrow \psi \rrbracket_\xi := (\vartheta_{n-1} \Rightarrow \vartheta_0, \ldots, \vartheta_{n-1} \Rightarrow \vartheta_0) \)
  - \( \llbracket \varphi \Leftrightarrow \psi \rrbracket_\xi := (\vartheta_{n-1} \Leftrightarrow \vartheta_0, \ldots, \vartheta_{n-1} \Leftrightarrow \vartheta_0) \)
• for explicit type converters, we define
  - \([\text{int2bv}(\tau)]_\xi := (b_{n-1}, \ldots, b_0)\) with \(\langle b_{n-1}, \ldots, b_0⟩ \gg 2 = [\tau]_\xi\) and \(n := \text{intsize}([n - 1])\) where \(\tau : \text{int}<n>\)
  - \([\text{nat2bv}(\tau)]_\xi := (b_{n-1}, \ldots, b_0)\) with \(\langle b_{n-1}, \ldots, b_0⟩ \gg 1 = [\tau]_\xi\) and \(n := \text{natsize}(\text{int}<n>\) where \(\tau : \text{nat}<n>\)
  - \([\text{bv2int}(\tau)]_\xi := (b_{n-1}, \ldots, b_0)\) with \(\langle b_{n-1}, \ldots, b_0⟩ \gg 2 = [\tau]_\xi\)
  - \([\text{bv2nat}(\tau)]_\xi := (b_{n-1}, \ldots, b_0)\) with \(\langle b_{n-1}, \ldots, b_0⟩ \gg 1 = [\tau]_\xi\)

• for arithmetic operators, we define\(^3\)
  - \([-\tau]_\xi := -[\tau]_\xi\)
  - \([\text{abs}(\tau)]_\xi := \lceil [\tau]_\xi \rceil\)
  - \([\tau + \pi]_\xi := [\tau]_\xi + [\pi]_\xi\)
  - \([\tau - \pi]_\xi := \begin{cases} 0 & \text{if } \tau, \pi : \text{nat}<n> \text{ or } \tau, \pi : \text{nat} \text{ and } [\tau]_\xi < [\pi]_\xi \\ [\tau]_\xi - [\pi]_\xi & \text{otherwise} \end{cases}\)
  - \([\tau \cdot \pi]_\xi := [\tau]_\xi \cdot [\pi]_\xi\)
  - \([\tau \div \pi]_\xi := \left[\frac{[\tau]_\xi}{[\pi]_\xi}\right]\)
  - \([\tau \mod \pi]_\xi := [\tau]_\xi \mod [\pi]_\xi\)
  - \([\text{exp2}(\tau)]_\xi := 2^\lceil [\tau]_\xi \rceil\)
  - \([\text{log2}(\tau)]_\xi := \left\lceil \log_2([\tau]_\xi) \right\rceil\)
  - \(\text{for } \tau : \text{nat}<m>, \text{ we define}\)
    \([\text{sat}<n>(\tau)]_\xi := \begin{cases} [\tau]_\xi & \text{if } 0 \leq [\tau]_\xi < [n] \\ [n-1] & \text{otherwise} \end{cases}\)
  - \(\text{for } \tau : \text{int}<m>, \text{ we define}\)
    \([\text{sat}<n>(\tau)]_\xi := \begin{cases} [\tau]_\xi & \text{if } -[n] \leq [\tau]_\xi < [n] \\ [n-1] & [\tau]_\xi \geq [n] \\ [-n] & [\tau]_\xi < -[n] \end{cases}\)

• for arithmetic relations, we define
  - \([\tau < \pi]_\xi := \langle [\tau]_\xi < [\pi]_\xi \rangle\)
  - \([\tau \leq \pi]_\xi := \langle [\tau]_\xi \leq [\pi]_\xi \rangle\)
  - \([\tau > \pi]_\xi := \langle [\tau]_\xi > [\pi]_\xi \rangle\)
  - \([\tau \geq \pi]_\xi := \langle [\tau]_\xi \geq [\pi]_\xi \rangle\)

• for equality and inequality operators, we define
  - \([\tau = \pi]_\xi := \langle [\tau]_\xi = [\pi]_\xi \rangle\)
  - \([\tau \neq \pi]_\xi := \langle [\tau]_\xi \neq [\pi]_\xi \rangle\)

• finally,

\(^3\)The arithmetic operators used to define the semantics are those that are defined for the set of integers \(\mathbb{Z}\). The type rules guarantee that \([\tau]_\xi \in \mathbb{N}\) or \([0, \ldots, n - 1]\) for unsigned types \(\tau\).
2.2 Expressions

- \( \llbracket (\varphi?\tau:\pi) \rrbracket_\xi \) := \( \begin{cases} \llbracket \varphi \rrbracket_\xi & \text{if} \ \llbracket \tau \rrbracket_\xi \\ \llbracket \pi \rrbracket_\xi & \text{otherwise} \end{cases} \)
- for \( \tau : \text{nat}<n> \), we define \( \llbracket \text{sizeOf}(\tau) \rrbracket_\xi := \text{natsize}(\llbracket n-1 \rrbracket) \)
- for \( \tau : \text{int}<n> \), we define \( \llbracket \text{sizeOf}(\tau) \rrbracket_\xi := \text{intsize}(\llbracket -n \rrbracket) \)

The semantics of variables, literals, and the tuple constructor and selector is clear. Recall that for every data type \( \alpha \), the semantics of the array type \( \text{array}(\alpha,n) \) is the set of functions that map the numbers \( \{0,\ldots,n-1\} \) to elements of the semantics of type \( \alpha \). Hence, the semantics of the array access is essentially the function application of the function \( \llbracket x \rrbracket_\xi \) to the argument \( \llbracket \tau \rrbracket_\xi \).

As we allow index expressions of type \( \text{int}<n> \), we included a modulo operation in the definition of the semantics, although the index expression should definitely have a nonnegative value. The reason for this is that we wish to allow expressions of type \( \text{int}<n> \) without using the \text{abs} operator to switch to a legal type whenever we know that the index expression is not negative. The alternative to use the absolute value of the index expression instead is not good for two reasons: first, the value \(-n\) will still lead to an overflow, and second the use of the modulo operator has additional interesting properties as will be explained below for bitvector operations. From a theoretical point of view, we could also require in rule (II.3) that the index expression has type \( \text{nat}<n> \) so that the programmer has to use the modulo operator in his or her own. However, we wish to remove that burden from the programmer.

Note that the only operation (except for packing in a tuple or selecting from a tuple) that can be applied to a variable of array type is to access one of the fields of the array. Therefore, only full array access is allowed, i.e., for a two dimensional array of integers it is not possible to access a row or column of that array (although this requires multiple applications of rule (II.3)).

Bitvector concatenation should be clear. Since we identify \( \text{bool} \) with \( \text{bv}[1] \), it is also possible to concatenate boolean values to bitvectors or to other boolean values.

For the explanation of the bitvector access operations, assume first that we have \( \langle \vartheta_{n-1},\ldots,\vartheta_0 \rangle := \llbracket \varphi \rrbracket_\xi \) and that \( p_1 := \llbracket \pi_1 \rrbracket \in \{0,\ldots,n-1\} \) and \( p_2 := \llbracket \pi_2 \rrbracket \in \{0,\ldots,n-1\} \). Under these assumptions, we have \( p_1 = (p_1 \mod n) \) and \( p_2 = (p_2 \mod n) \), and therefore the following holds:

- \( \varphi\{\pi_1\} \) represents \( \vartheta_{p_1} \), i.e., the \( p_1 \)-th bit
- \( \varphi\{\pi_1 : \pi_2\} \) represents \( \langle \vartheta_{p_1},\ldots,\vartheta_{p_2} \rangle \) of length \( p_1 - (p_2 - 1) \)
- \( \varphi\{\pi_1 :\} \) represents the suffix \( \langle \vartheta_{p_1},\vartheta_{p_1-1},\ldots,\vartheta_0 \rangle \) of length \( p_1 + 1 \)
- \( \varphi\{:: \pi_2\} \) represents the prefix \( \langle \vartheta_{n-1},\vartheta_{n-2},\ldots,\vartheta_{p_2} \rangle \) of length \( n - p_2 \)

Now assume that \( -p_1 = \llbracket \pi_1 \rrbracket \) and \( -p_2 = \llbracket \pi_2 \rrbracket \) are negative (hence, \( 0 < p_1,p_2 < n \)). Then, we have \( \llbracket \pi_1 \rrbracket \mod n = n-p_1 \) (since \( -p_1 = (-1)\cdot n + (n - p_1) \), thus \( -1 = ((-p_1) \div n) \) and \( n - p_1 = ((-p_1) \mod n) \)). Hence, we have:

- \( \varphi\{\pi_1\} \) represents \( \vartheta_{n-p_1} \), i.e., the \( n-p_1 \)-th bit
• \( \varphi\{\pi_1 : \pi_2\} \) represents \((\vartheta_{n-p_1}, \ldots, \vartheta_{n-p_2})\)

• \( \varphi\{\pi_1 : \} \) represents the suffix \((\vartheta_{n-p_1}, \vartheta_{n-p_1-1}, \ldots, \vartheta_0)\) of length \(n - p_1 + 1\)

• \( \varphi : \pi_2 \) represents the prefix \((\vartheta_{n-1}, \vartheta_{n-2}, \ldots, \vartheta_{n-p_2})\) of length \(p_2\)

In particular, \( \varphi\{-1\} \) is the leftmost bit while \( \varphi\{0\} \) is the rightmost bit. Moreover, \( \varphi\{ : \pi_2\} \) represents the prefix with \(p_2\) bits and \( \varphi\{\pi_1 : \} \) represents the suffix with \(p_1 + 1\) bits. Note that \( \varphi\{\pi_1 : \pi_1\} \) represents \((\vartheta_{p_1})\), i.e., the bitvector that only contains the \(p_1\)-th bit, while \( \varphi\{\pi_1\} \) is only the \(p_1\)-th bit, i.e., a boolean value. However, since we do not distinguish between \(\text{bool}\) with \(\text{bv}[1]\), this does not matter and both values are the same.

Note that the expression \( \varphi\{-1\} \) for accessing the leftmost bit is much more convenient than the equivalent expression \( \varphi\{\text{sizeOf}(\varphi) - 1\} \) that would have to be used without the freedom to use index expressions of type \(\text{int}^{<n>}\) and the semantics with the modulo operation.

For the slicing expressions \( \varphi\{\pi_1 : \pi_2\} \), it is checked at compile time whether \( p_1 := (\|\pi_1\| \mod n) \) is greater than or equal to \( p_2 := (\|\pi_2\| \mod n) \). If this is not the case, a type error is reported at compile time (note that \(\pi_1\) and \(\pi_2\) are static expressions!).

Finally, the replicate operator \( :: \) is used to generate a bitvector of specified length that contains the same bits. For example, \{\(b::5\)\} with a boolean expression \(b\) generates a bitvector of type \(\text{bv}[5]\) that contains the bit \(b\) five times. Together with the \(\oplus\) operator, this can be used to define many macro operators like zero or sign extension as shown in Section 2.2.5.

The fromArray operator can be used to convert a boolean array to a bitvector. Note how the bits of the resulting bitvector are arranged. Last but not least, reverse reverses the bits of a given bitvector, i.e., \(\text{reverse}(b)\) is represented by \((b_0, \ldots, b_{n-1})\).

The semantics of the (bitwise) boolean operations is very simple. For reasons of completeness, we define the operators we used to define the semantics in the following table:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(\neg x)</th>
<th>(x \land y)</th>
<th>(x \lor y)</th>
<th>(x \rightarrow y)</th>
<th>(x \leftrightarrow y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

For conjunction and disjunction, there are also generic variants that are unrolled during compilation to normal conjunctions and disjunctions, respectively. For this reason, it is required that the index bounds must be static expressions. Examples for these expressions are as follows (more explanations on the generic syntax is also found in Section 3.4.7):

• \(\forall (\text{nat}<4> \ i=0u..3u) \ a[i] \) and \(\neg b[i]\)

• \(\exists (\text{nat}<4> \ i=0u..3u) \ a[i] \) and \(b[i]\)

In case, that the bounds specify an empty set of indices, then the \(\forall\) construct evaluates to true and the \(\exists\) expression evaluates to false.
2.2 Expressions

The semantics of an arithmetic expression is simply that number that is obtained by applying the corresponding arithmetic operation to the numbers that are obtained by evaluating the operand expressions. In a similar way, we define the semantics of comparison operators and the equality operator. Precise definitions with possible implementations are given in Appendix B.

Further comments are necessary for the semantics of division and modulo operators, since there is still no standard definition in computer science. In general, the quotient \( q \) and the remainder \( r \) obtained by the division of integer \( a \) by integer \( b \) should always fulfill (1) \( a = q \cdot b + r \) and (2) \( |r| < |b| \) (since this is already required by Euclidean rings). However, these requirements do not uniquely determine \( q \) and \( r \) for given numbers \( a \) and \( b \): Indeed, different additional conditions are used by different programming languages and microprocessors to select one of these two possible solutions (see Figure H.6 in [110]). In particular, the following definitions can be used:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( 0 \leq r )</th>
<th>( \text{round } q ) to 0</th>
<th>( \text{round } q ) to (-\infty)</th>
<th>( \text{round } q ) to (+\infty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x/y \times % y )</td>
<td>( x/y \times % y )</td>
<td>( x/y \times % y )</td>
<td>( x/y \times % y )</td>
<td>( x/y \times % y )</td>
<td>( x/y \times % y )</td>
</tr>
<tr>
<td>5 3</td>
<td>1 2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5 -3</td>
<td>-1 2</td>
<td>-1</td>
<td>2</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>-5 3</td>
<td>-2 1</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-5 -3</td>
<td>2 1</td>
<td>1</td>
<td>-2</td>
<td>1</td>
<td>-2</td>
</tr>
</tbody>
</table>

However, note that the conditions (1) and (2) given above hold in all of these variants. Moreover, the above four variants are equally expressive, since we can compute one out of the other. To this end, let us first define the above four variants in terms of unsigned division: Given integers \( x \) and \( y \), let us first compute \( q := |x| / |y| \) and \( r := |x| \mod |y| \) for the following definitions:

- \( (q_Q, r_Q) := \begin{cases} (q, r) & \text{if } x < 0 \land y < 0 \land r = 0 \\ (q + 1, |y| - r) & \text{if } x < 0 \land y < 0 \land r > 0 \\ (-q, r) & \text{if } x < 0 \land y > 0 \land r = 0 \\ (-q + 1, |y| - r) & \text{if } x < 0 \land y > 0 \land r > 0 \\ (-q, r) & \text{if } 0 \leq x \land y < 0 \\ (q, r) & \text{if } 0 \leq x \land 0 < y \end{cases} \)

- \( (q_0, r_0) := \begin{cases} (q, r) & \text{if } x < 0 \land y < 0 \\ (-q, r) & \text{if } x < 0 \land 0 < y \end{cases} \)

- \( (q_{-\infty}, r_{-\infty}) := \begin{cases} (q, r) & \text{if } x < 0 \land y < 0 \\ (-q + 1, -(r - |y|)) & \text{if } x < 0 \land 0 < y \\ (-q + 1, (r - |y|)) & \text{if } 0 \leq x \land y < 0 \\ (q, r) & \text{if } 0 \leq x \land 0 < y \end{cases} \)

- \( (q_{+\infty}, r_{+\infty}) := \begin{cases} (q + 1, -(r - |y|)) & \text{if } x < 0 \land y < 0 \\ (-q, r) & \text{if } x < 0 \land 0 < y \end{cases} \)
Based on the above definitions, we can prove that the following equations hold that can be used to implement the other kinds of division operations of the one that has been chosen in Quartz:

- \((q_0, r_0) := \begin{cases} 
(-q_Q, r_Q - 1) & \text{if } x < 0 \land y < 0 \\
(-q_Q, r_Q + 1) & \text{if } x < 0 \land 0 < y \\
(q_Q, r_Q) & \text{if } 0 \leq x \land y < 0 \\
(q_Q, r_Q) & \text{if } 0 \leq x \land 0 < y 
\end{cases}\)

- \((q_{-\infty}, r_{-\infty}) := \begin{cases} 
(q_Q - 1, r_Q - |y|) & \text{if } x < 0 \land y < 0 \\
(q_Q, r_Q) & \text{if } x < 0 \land 0 < y \\
(q_Q - 1, r_Q - |y|) & \text{if } 0 \leq x \land y < 0 \\
(q_Q, r_Q) & \text{if } 0 \leq x \land 0 < y 
\end{cases}\)

- \((q_{+\infty}, r_{+\infty}) := \begin{cases} 
(q_Q + 1, r_Q - |y|) & \text{if } x < 0 \land 0 < y \\
(q_Q, r_Q) & \text{if } 0 \leq x \land y < 0 \\
(q_Q + 1, r_Q - |y|) & \text{if } 0 \leq x \land 0 < y 
\end{cases}\)

In Quartz, the division and modulo operations are defined such that the remainder is always non-negative, i.e., for \(y \neq 0\), \(x / y\) and \(x \% y\) are the uniquely determined numbers that satisfy the following conditions (\(\text{abs}(y)\) denotes the absolute value of \(y\)):

\[x = (x / y) * y + (x \% y)\] and \(0 < x \% y\) and \(x < \text{abs}(y)\)

Hence, the remainder \(x \% y\) is always a non-negative number and therefore its type is \(\text{nat}<...>\). It is moreover easily seen that the remainder \(x \% y\) requires one bit less than the divisor \(y\).

From a mathematical point of view, division by zero and subtraction \(x - y\) with \(x < y\) for unsigned numbers are not defined. In many programming languages, such cases lead to runtime errors that have to be caught at runtime. In a similar way, some hardware circuits like microprocessors know the concept of exceptions and are able to react to these situations in a corresponding way. In future versions of Quartz, we plan to implement an exception handling mechanism similar to Java to handle these cases in a consistent setting.

### 2.2.4 Static (i.e. Compile-time Constant) Expressions

Quartz is a statically typed language, i.e., the types of all expressions are determined at compile time and do not change during runtime. This means that an expression has a certain type like \(\text{nat}<6>\) even if its current value may only be 2. In contrast, in a dynamically typed language, the type would be adapted to the current value, so that the type of a variable with value 2 would be rather \(\text{nat}<3>\) than \(\text{nat}<6>\). Dynamic typing is however not reasonable for hardware description languages, and therefore a statically typed approach is preferable for Quartz, too.

Although Quartz is a statically typed language, there is one minor exception: the evaluation of static expressions is done at compile time, and the type
of the static expression is the type of the finally obtained constant. Hence, the type of \(3\times 7-14\) is \(\text{int}<8>\) since the expression evaluates to the constant 7 of type \(\text{int}<8>\). In contrast, the type of \(x\times y-z\) with variables \(x, y,\) and \(z\) of types \(\text{int}<4>, \text{int}<8>,\) and \(\text{int}<15>\) is \(\text{int}<48>\). This does not really violate the static typing rules if we interpret the static evaluation as a step that is done before type checking.

In principle, only variable-free expressions are static expressions with the exception that also all expressions of the form sizeOf(\(\tau\)) are static expressions. This is due to the fact that the language is statically typed and that therefore the bitwidth of an expression \(\tau\) does not change during runtime.

### 2.2.5 User-Defined Functions

Clearly, many more operators can be thought of, and in practice many more are really required to achieve some degree of convenience. Instead of adding too many operators to the core language, we prefer to define further operators as simple macros that can be added by preprocessor directives. The following list contains some of the most popular macro operators whose meaning is given by the semantics of their definition:

- \#define odd(x) \((x \% 2u)==1u\)
- \#define even(x) \((x \% 2u)==0u\)
- \#define max(x,y) \((x<y?y:x)\)
- \#define min(x,y) \((x<y?x:y)\)
- \#define shtr(b,i) \{false::i\}@b{:i}
- \#define shtl(b,i) b{-(i+1):}@{false::i}
- \#define rotr(b,i) b{(i-1):}@b{:i}
- \#define rotl(b,i) b{-(i+1):}@b{-(i-1)}
- \#define zeroext(b,n) \{false::n-sizeOf(b)\}@b
- \#define signext(b,n) \{b{sizeOf(b)-1}::n-sizeOf(b)\}@b
- \#define hd(b) b{-1}
- \#define tl(b) b{-2:}
- \#define last(b) b{0}
- \#define butlast(b) b{:1}
- \#define prefix(b,k) b{:-(k-1)}
- \#define suffix(b,k) b{-(k-1)}

### 2.3 Specifications

The Quartz language allows to implement systems with a complex temporal behavior in software and hardware independent of the underlying technology like the processor architecture. It is therefore a high-level programming language that is preferable in early design phases. In these early design phases, the formal verification of the system is a task that is gaining more
Data Types, Expressions, and Specifications

and more importance. This is not only due to the increasing use of embedded systems in safety-critical applications, but also due to economic reasons: The currently used ASIC technology is so expensive that design errors that are found after the layout synthesis can not be afforded. Moreover, upcoming System-on-a-Chip design flows challenge the reuse of available hardware circuits (IP-blocks). For this reason, these reused hardware circuits are fabricated in much more products than before and therefore the correctness is absolutely mandatory and it can be afforded.

For this reason, a major aspect of Quartz are built-in capabilities that support the formal verification of the implemented modules. Quartz offers a wide spectrum of specification logics [225] that can be used for this purpose, including LTL, CTL, LeftCTL*, and an extended variant of the full µ-calculus even with past modal operators. In the following, we briefly describe the syntax, type rules, and the semantics of specifications. We can not go into details on the semantics of these specification logics and refer instead to [225] for this purpose.

2.3.1 Syntax

Table 2.2 shows the operators that can be used to construct specifications. There are two special constants InitialStates and ReachableStates that are often required in specifications. While ReachableStates can be easily defined as a macro for nu x. InitialStates or <::>x, it is not possible to derive a description of the initial states. Note that 'state' refers here to the labeled transition system that is derived from the program. Theses states include not only the program’s control state and its local and output variables, but also the current input variables. Therefore, even if the program has only one initial state, the corresponding labeled transition system may have many copies of that which are endowed with additional inputs.

The meaning of the boolean operators should be clear. The next group of operators are linear time temporal operators that are categorized into future time and past time temporal operators. It is known that past time temporal operators can be eliminated [104, 152, 225], so we could theoretically ignore them. However, first, it turned out that past time temporal operators can be used to construct exponentially more succinct formulas to express the same property with only future time temporal operators [173]. Second, it has been realized that in many cases the use of past time temporal operators leads to much more readable specification. Hence, we feel that the use of past time temporal operators is very important for practice.

The path quantifiers E and A that are used to quantify whether a path property holds on at least one of the computations starting in a state or on all of its computations. Using these quantifiers one can specify that from the current state, there will be a possible further computation that satisfies a given path property of that all further computations starting in this state satisfy a
given path property. Note that these operators perform a certain reset of time that is respected by the temporal operators.

The next two groups of operators refer to the modal $\mu$-calculus. The modal operators $<>$ and $[]$ mean that a property holds on at least one of the successor states or on all (if there are any) successor states. The reversed variants $:<>$ and $[:]$ check the same for the predecessor states. Finally, the operators $\text{mu}$ and $\text{nu}$ are used to specify least and greatest fixpoints as defined in the $\mu$-calculus.
There is also the possibility to make use of the vectorized $\mu$-calculus which is more succinct than the flat $\mu$-calculus. An example specification looks as follows:

```
fixpoints {
    mu x1 = y & alpha1 | beta & <>x1;
    mu x2 = y & alpha2 | beta & <>x2;
    nu y = beta & <>x1 & <>x2;
} in y
```

Vectorized $\mu$-calculus expressions can alternatively be read as alternating tree automata, so that Averest's specification logic is also capable to handle these specifications.

In such a vectorized $\mu$-calculus expression, the upper equations refer to inner fixpoints and the lower ones to outer fixpoints. In the above case, we could therefore use the following equivalent flat $\mu$-calculus expression:

```
nu y. (beta &
    <> nu x1. (y & alpha1 | beta & <>x1) &
    <> nu x2. (y & alpha2 | beta & <>x2))
```

Finally, *exists* $x: \alpha. \varphi$ and *forall* $x: \alpha. \varphi$ are quantified expressions that state that a property $\varphi$ holds for one or all instances of a variable $x$ of type $\alpha$. The generic conjunction/disjunction are obviously finite cases of quantifications.

The precedences and associativities of the boolean operators have already been given for program expressions and hold also for specifications. For the other operators, the following rules apply:

- Monadic operators bind stronger than other operators. Hence, $X a & b$ is parsed as $(X a) & b$.
- Quantifiers and fixpoint expressions (both are sometimes called binders) are also viewed as unary operators. Hence, $forall x. 2<x & x<10$ is parsed as $(forall x. 2<x) & x<10$, and the programmer has to use delimiters for the other case $(forall x. (2<x & x<10))$.
- The binary temporal operators have weaker priority than the operators used in expressions of statements. It is required to always use delimiters $[\ ]$ around the binary temporal operators, e.g., we have to write $[\varphi \ SU \ \psi]$.

---

4 The property holds on states that have at least one path where all the time $beta$ holds, and infinitely often $alpha1$ and infinitely often $alpha2$ hold. It can therefore be used to check the acceptance of generalized Büchi automata.
2.3 Specifications

2.3.2 Type Rules

For the definition of type rules for the specifications, we have to distinguish between state and path formulas. State formulas express a property concerning the state of the program, i.e., state formulas refer to a macro step of the program. In contrast, path formulas express a property of a computation trace that consists of infinitely many states: Even if the program terminates, the semantics of statements that we will define in the next section is defined so that there will be a self-loop in the final state. This is very important for the definition of temporal operators, so that properties like ‘infinitely often’ are not questionable.
### Temporal future operators

(1) \( \varphi : \text{pathf} \quad X \varphi : \text{pathf} \)

(2) \( \varphi : \text{pathf} \quad G \varphi : \text{pathf} \)

(3) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad F \varphi : \text{pathf} \)

(4) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ SU \ \psi ] : \text{pathf} \)

(5) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ WU \ \psi ] : \text{pathf} \)

(6) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ SB \ \psi ] : \text{pathf} \)

(7) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ WB \ \psi ] : \text{pathf} \)

(8) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ SW \ \psi ] : \text{pathf} \)

(9) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ WW \ \psi ] : \text{pathf} \)

### Temporal past operators

(10) \( \varphi : \text{pathf} \quad \text{PSX} \ \varphi : \text{pathf} \)

(11) \( \varphi : \text{pathf} \quad \text{PWX} \ \varphi : \text{pathf} \)

(12) \( \varphi : \text{pathf} \quad \text{PG} \ \varphi : \text{pathf} \)

(13) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \text{PF} \ \varphi : \text{pathf} \)

(14) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PSU \ \psi ] : \text{pathf} \)

(15) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PWU \ \psi ] : \text{pathf} \)

(16) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PSB \ \psi ] : \text{pathf} \)

(17) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PWB \ \psi ] : \text{pathf} \)

(18) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PSW \ \psi ] : \text{pathf} \)

(19) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad [\varphi \ PWW \ \psi ] : \text{pathf} \)

### Boolean closure

(20) \( \varphi : \text{pathf} \quad \neg \varphi : \text{pathf} \)

(21) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \varphi \mathbf{x o r} \ \psi : \text{pathf} \)

(22) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \varphi \mathbf{\&} \ \psi : \text{pathf} \)

(23) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \varphi \mathbf{|} \ \psi : \text{pathf} \)

(24) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \varphi \mathbf{\rightarrow} \ \psi : \text{pathf} \)

(25) \( \varphi : \text{pathf} \quad \psi : \text{pathf} \quad \varphi \mathbf{\leftrightarrow} \ \psi : \text{pathf} \)

---

Fig. 2.5. Type System for Path Formulas

The type rules given in Figures 2.4 and 2.5 show the type rules for specifications where statef and pathf denote state and path formulas, respectively.

Rule (S.1) and (S.2) state that InitialStates and ReachableStates are state formulas. (S.3) states that boolean program expressions are state formulas, and so are boolean combinations of state formulas which is formalized with rules (S.15)-(S.20). Every state formula is a path formula, which
is expressed by rule (S.4). The other rules of Figure 2.4 simply define the
typical state formulas (see any references for temporal logics, as for exam-
ple [70, 225]). Note that in the rules of the quantifiers, the bound variable \(x\) describes a set of states.

Figure 2.5 describes the type rules for path formulas which are very sim-
ple. It is only stated that the temporal future and past time operators expect
path formulas are arguments and that they generate a new path formula. We
have to repeat again the boolean operators in rules (P.20)-(P.25), this time
for path formulas.

2.3.3 Semantics

The semantics of specifications is formally defined on a labeled transition sys-
tem that is derived from a Quartz program. In principle, the labeled transition
system is the semantics of the Quartz program, since its states and transitions
 correspond with the reachable states and macro steps of the program. The
states are labeled with a variable assignment, and we moreover, classify the
variables into uncontrollable input, controllable input and output variables.

**Definition 2.6 (Kripke Structures).** A Kripke structure \(K = (I, S, R, \mathcal{L}, \mathcal{V})\)
consists of a set of states \(S\), its initial states \(I \subseteq S\), and a set of transitions
\(R \subseteq S \times S\). Moreover, the set of variables \(\mathcal{V}\) is partitioned into uncontrollable
input variables \(\mathcal{V}_I\), controllable input variables \(\mathcal{V}_C\), and state variables \(\mathcal{V}_S\). The
labeling function \(\mathcal{L}\) maps every state \(s \in S\) to a variable assignment \(\mathcal{L}(s)\) that
maps every variable to a value that is consistent with its type.

We will describe in the hardware synthesis chapter in detail how a labeled
transition system as described above is obtained from a Quartz program.

To explain the semantics of specifications, we have to introduce some aux-
iliary definitions in advance. The first is the notion of a path through the
Kripke structure, which is informally clear. Formally, a path \(\pi : \mathbb{N} \rightarrow S\)
is a function that maps natural numbers to states of the structure so that
\((\pi(t), \pi(t+1)) \in R\) holds. Note that paths are by definition infinitely long. We
say that the path starts in state \(\pi(0)\) which need not necessarily be an initial
state. The set of paths of a Kripke structure \(K\) that start in a state \(s \in S\) is
written as \(\text{Paths}_K(s)\)

The second auxiliary definition is that of a modified Kripke structure. This
means that we keep the states and transitions of the structure, but modify its
labels as specified by a set of states \(Q\) and a boolean variable \(x\):

**Definition 2.7 (Kripke Structures).** For a Kripke structure \(K = (I, S, R, \mathcal{L}, \mathcal{V})\),
a variable \(x \in \mathcal{V}\) and a set of states \(Q \subseteq S\), we define the structure \(K^Q_x =
(I, S, R, \mathcal{L}', \mathcal{V})\) with (where \(\equiv\) denotes syntactic equality):

\[
\mathcal{L}'(s)(y) = \begin{cases} 
\mathcal{L}(s)(y) & \text{if } y \not\equiv x \\
\mathcal{L}(s)(y) & \text{if } y \equiv x
\end{cases}
\]
Having all the ingredients, we can now present the definition of the semantics of the single operators.

**Definition 2.8 (Semantics of Specifications).** Given a path $\pi$ of a Kripke structure $\mathcal{K} = (I, S, R, L, V)$, and a number $t \in \mathbb{N}$, the following rules define the semantics of path formulas:

**atomic path formulas**
- $(K, \pi, t) \models \varphi$ iff $(K, \pi^{(t)}) \models \varphi$ for each state formula $\varphi$

**boolean closure**
- $(K, \pi, t) \models !\varphi$ iff not $(K, \pi, t) \models \varphi$
- $(K, \pi, t) \models \varphi \& \psi$ iff $(K, \pi, t) \models \varphi$ and $(K, \pi, t) \models \psi$
- $(K, \pi, t) \models \varphi \lor \psi$ iff $(K, \pi, t) \models \varphi$ or $(K, \pi, t) \models \psi$
- $(K, \pi, t) \models \varphi \rightarrow \psi$ iff $(K, \pi, t) \models \varphi$ implies $(K, \pi, t) \models \psi$
- $(K, \pi, t) \models \varphi < \rightarrow \psi$ iff $(K, \pi, t) \models \varphi$ is equivalent to $(K, \pi, t) \models \psi$
- $(K, \pi, t) \models \varphi \land \psi$ iff $(K, \pi, t) \models \varphi \land \psi$

**temporal future operators**
- $(K, \pi, t) \models X\varphi$ iff $(K, \pi, t + 1) \models \varphi$
- $(K, \pi, t) \models G\varphi$ iff for all $\delta$, we have $(K, \pi, t + \delta) \models \varphi$
- $(K, \pi, t) \models F\varphi$ iff for at least one $\delta$, we have $(K, \pi, t + \delta) \models \varphi$
- $(K, \pi, t) \models [\varphi SU \psi]$ iff there is a $\delta \geq t$ such that $(K, \pi, \delta) \models \psi$ holds and for all $x$ with $t \leq x < \delta$, we have $(K, \pi, x) \models \varphi$
- $(K, \pi, t) \models [\varphi SB \psi]$ iff there is a $\delta \geq t$ such that $(K, \pi, \delta) \models \varphi$ holds and for all $x$ with $t \leq x \leq \delta$, we have $(K, \pi, x) \models \psi$
- $(K, \pi, t) \models [\varphi SW \psi]$ iff there is a $\delta \geq t$ such that $(K, \pi, \delta) \models \varphi \land \psi$ holds and for all $x$ with $t \leq x < \delta$, we have $(K, \pi, x) \models !\varphi$
- $(K, \pi, t) \models [\varphi WU \psi]$ iff $(K, \pi, t) \models [\varphi SU \psi] \parallel G\varphi$
- $(K, \pi, t) \models [\varphi WW \psi]$ iff $(K, \pi, t) \models [\varphi SW \psi] \parallel G!\psi$
- $(K, \pi, t) \models [\varphi WB \psi]$ iff $(K, \pi, t) \models [\varphi SB \psi] \parallel G!\psi$

**temporal past operators**
- $(K, \pi, t) \models PWX\varphi$ iff $t = 0$ or $(K, \pi, t - 1) \models \varphi$
- $(K, \pi, t) \models PSX\varphi$ iff $t > 0$ and $(K, \pi, t - 1) \models \varphi$
- $(K, \pi, t) \models PG\varphi$ iff for all $\delta \leq t$, we have $(K, \pi, \delta) \models \varphi$
- $(K, \pi, t) \models PF\varphi$ iff for at least one $\delta \leq t$, we have $(K, \pi, \delta) \models \varphi$
- $(K, \pi, t) \models [\varphi PSU \psi]$ iff there is a $\delta \leq t$ such that $(K, \pi, \delta) \models \psi$ holds and for all $x$ with $\delta < x \leq t$, we have $(K, \pi, x) \models \varphi$
- $(K, \pi, t) \models [\varphi PSB \psi]$ iff there is a $\delta \leq t$ such that $(K, \pi, \delta) \models \varphi$ holds and for all $x$ with $\delta \leq x \leq t$, we have $(K, \pi, x) \models !\psi$
- $(K, \pi, t) \models [\varphi PSW \psi]$ iff there is a $\delta \leq t$ such that $(K, \pi, \delta) \models \varphi$ holds and for all $x$ with $\delta < x \leq t$, we have $(K, \pi, x) \models !\psi$
- $(K, \pi, t) \models [\varphi PWU \psi]$ iff $(K, \pi, t) \models [\varphi PSU \psi] \parallel G\varphi$
- $(K, \pi, t) \models [\varphi PW \psi]$ iff $(K, \pi, t) \models [\varphi PSW \psi] \parallel G!\psi$
- $(K, \pi, t) \models [\varphi PWB \psi]$ iff $(K, \pi, t) \models [\varphi PSB \psi] \parallel G!\psi$

For a given state $s$ of a structure $\mathcal{K} = (I, S, R, L, V)$, the semantics of a state formula is given by the following definitions:
atomic state formulas

- \((K, s) \models \text{InitialStates} \iff s \in \mathcal{I}\)
- \((K, s) \models \text{ReachableStates} \iff (K, s) \models \nu x. \text{InitialStates} \land <>x\)
- \((K, s) \models \varphi \iff J^\varphi K L^s = \text{true}\) for all boolean program expressions \(\varphi\)

boolean closure

- \((K, s) \models \neg \varphi \iff (K, s) \not\models \varphi\)
- \((K, s) \models \varphi \land \psi \iff (K, s) \models \varphi \land (K, s) \models \psi\)
- \((K, s) \models \varphi \lor \psi \iff (K, s) \models \varphi \lor (K, s) \models \psi\)
- \((K, s) \models \varphi \implies \psi \iff (K, s) \models \varphi \implies (K, s) \models \psi\)
- \((K, s) \models \varphi \iff (K, s) \models \psi \iff (K, s) \models \neg \psi\)

path quantifiers

- \((K, s) \models \text{E}\varphi \iff \text{there is a path } \pi \in \text{Paths}_K(s) \text{ such that } (K, \pi, 0) \models \varphi\)
- \((K, s) \models \text{A}\varphi \iff \text{for all paths } \pi \in \text{Paths}_K(s) \text{ we have } (K, \pi, 0) \models \varphi\)

modal operators

- \((K, s) \models <>\varphi \iff \text{there is a } s' \in S \text{ with } (s, s') \in \mathcal{R} \text{ and } (K, s') \models \varphi\)
- \((K, s) \models []\varphi \iff \text{for all } s' \in S \text{ with } (s, s') \in \mathcal{R}, \text{ we have } (K, s') \models \varphi\)
- \((K, s) \models <>\varphi \iff \text{there is a } s' \in S \text{ with } (s', s) \in \mathcal{R} \text{ and } (K, s') \models \varphi\)
- \((K, s) \models []\varphi \iff \text{for all } s' \in S \text{ with } (s', s) \in \mathcal{R}, \text{ we have } (K, s') \models \varphi\)

quantifiers

- \((K, s) \models \exists x. \varphi \iff \text{there is a } Q \subseteq S \text{ such that } (K^Q, s) \models \varphi\)
- \((K, s) \models \forall x. \varphi \iff \text{for all } Q \subseteq S \text{ we have } (K^Q, s) \models \varphi\)
- \((K, s) \models \nu x. \varphi \iff s \in \hat{Q}, \text{ where } \hat{Q} \text{ is the least set of states } Q \subseteq S \text{ that satisfies the equation } Q = \{ s' \in S \mid (K^Q, s') \models \varphi \}\)
- \((K, s) \models \nu x. \varphi \iff s \in \check{Q}, \text{ where } \check{Q} \text{ is the greatest set of states } Q \subseteq S \text{ that satisfies the equation } Q = \{ s' \in S \mid (K^Q, s') \models \varphi \}\)
In the previous chapter, we have described the syntax and semantics of types, program expressions, and specifications. The building blocks to model a hardware-software system are however modules that are described in this chapter, and their formal semantics is described in the following chapter.

Systems modelled in Quartz can be organized in different files, where each file consists of a list of modules. The semantics of a file is thereby the semantics of the first module in the file. The remaining modules in the file are either used in the first module or they are dead code. Every module has a name and the names in one file must be pairwise distinct. If a system is described in several files, then the names of the first modules in these files must also be pairwise different, but the other modules may share the same names. It is a good practice to use the same name for the files and their first module and to place the files that describe one system in a common directory.

There are two kinds of modules: behavioral and specification modules. Specification modules are simply a list of specifications, while behavioral modules contain the model of the system under consideration. In the following, we concentrate on behavioral modules.

Modules are viewed similar to hardware circuits in that they describe systems that read some data streams and generate some other data streams. The data streams that are read are the inputs and the data streams that are generated are the output data streams. Since Quartz is a synchronous language, the data streams have an underlying discrete time, i.e., they are infinite sequences of values, i.e., essentially functions of type $\mathbb{N} \rightarrow \alpha$ for some set of values $\alpha$. The temporal behavior of a module is divided into discrete reactions that are also called macro steps. In each reaction or macro step, the module reads one data value from all input streams and generates one data value for all output streams. The computation of the output values from the input values depends on an internal state of the module that is also updated in a reaction step. To be more precise, all variables have one and only one value per reaction step, and also the internal state of a module is uniquely defined in one reaction step. Depending on the current input values and the current internal state,
the reaction of the module determines therefore determines (1) the current output values and (2) the next internal state for the next reaction step.

This is in total accordance to synchronous hardware circuits that share the same underlying execution model. In contrast to synchronous hardware circuits, the reaction steps of a synchronous language like Quartz need not necessarily be equidistant, i.e., different reaction steps may required a different amount of real time. Synchronous language like Quartz do however not deal with physical time and replace time with a more abstract notion of reactions that are viewed as 'logical time'. Clearly, it is nevertheless possible to translate a synchronous program to a synchronous hardware circuit which is much clearer and simpler than for the still prevailing hardware description languages Verilog and VHDL with their discrete-event based notion of time (see Chapter 6). The frequency of the clock depends then, as it is always the case for synchronous hardware circuits, on the length of the longest reaction step.

If software synthesis is intended, then the length of the longest reaction step determines again the worst case reaction time. Note that an estimation in terms of physical time is only possible when the final processor architecture and even more the microprocessor itself together with the assembly code is available. Nevertheless, the use of synchronous languages still offers many advantages for the design of software systems that have to fulfill given real-time constraints. The main reason for this is that the number of machine instructions that can be executed in one reaction step is finite, since synchronous languages do not allow data-dependent loops in their reaction steps. For this reason, the finitely many actions of a reaction steps can be mapped to finitely many machine instructions so that tight estimates of the worst case reaction time can be obtained.

In all cases, i.e., whether a later software or hardware synthesis is intended, good synchronous programs should have balanced reactions, i.e., the reactions should have approximately equal execution times which is quite independent of the final realization in hardware or software. However, this is not a requirement for correct programs, instead, it is rather an issue of optimization, and therefore we will ignore such issues in the following. For the efficiency of synthesized systems, this issue is however very important and may require to transform an initial system by methods like pipelining or retiming [25, 95, 134, 167, 168, 237].

In this Chapter, we describe the syntax of modules which includes the syntax of interfaces and the module's body statement that determines how the outputs and the next internal state is computed from the current inputs. We therefore start with the declaration of module interfaces and proceed with the definition of statements. To this end, we distinguish between core statements and macro statements. Core statements are used in later chapters to explain the hardware and software synthesis while macro statements can be easily defined in terms of core statements. However, we do not insist to have a minimal set of core statements, and aim at finding a good compromise be-
tween a relatively small set of core statements and simple definitions of macro statements.

3.1 Modules and Interface Declarations

3.1.1 Interface Declarations

Besides the name, each module has an interface that defines the input and output data streams where values are read from and values are written to in each reaction step. The computation of the output values and the next internal state is determined by the body statement that is considered in the next section. The body statement must therefore know the names, types and further information on the input and output variables. This information is provided by the declaration of variables in the module’s interface.

The interface consists of the declaration of the variables that hold in each reaction step the values that are read from input streams and that are written to output streams. In addition to the names, the interface declaration provides also for all variables the types, data flow directions, and storage classes. The available types have already been described in detail in the previous chapter, so that we only have to add how types are used in the declaration.

Concerning the dataflow direction, variables declared in the interface are classified into (uncontrollable) inputs, controllable inputs, and outputs of the module. To this end, the names of the variables are possibly prefixed as follows:

- Uncontrollable input variables are not prefixed.
- Controllable input variables are prefixed with ?.
- Output variables are prefixed with &.

The body statement of the module can only determine the value of output variables in each reaction step. The values of input variables, regardless whether they are controllable or not, are determined by the environment. The difference between controllable and uncontrollable variables is that controllable variables are viewed to be under control of the system, but not by the current module, whether uncontrollable variables are neither under control of the system nor by the module. The code generation and the semantics of modules and statements does not distinguish between controllable and uncontrollable variables. This distinction is used for applications of supervisory control, where it can be checked whether for all uncontrollable input streams, there are controllable input streams and output streams such that a given specification holds. This is important for debugging where these questions can help to determine a modulo that can be changed so that a desired specification will finally hold. Supervisory control is a formal method that is a generalization of formal verification.
Finally, declared variables have either one of the storage classes *event* or *memorized*, which is not relevant for input variables. The storage class determines the semantics of the generated output data streams as follows: Memorized output variables store their current value unless it is changed by an action of the module. Event output variables do not store their current value on their own, and instead are reset to a default value if the module does not provide a new value in the considered reaction step. For this reason, one may view input variables having by default the event storage class.

The classification of variables into *event* or *memorized* ones is very important and motivated by hardware circuits: outputs of combinational gates are not memorized and therefore, these can be modeled with output variables of event storage class. In contrast, outputs of registers are memorized and therefore correspond with the memorized storage class. Programming languages used for the implementation of software do only know memorized variables whose default use is for hardware circuit synthesis however too expensive. The same holds for their use in formal verification methods like model checking, where memorized output variables lead to state variables of the considered automaton or Kripke structure, while event output variables may be simply defined in terms of the state variables. The compiler can easily determine whether it is possible to define an event variable in terms of other variables as will be explained in Chapter 6.

**Behavioral Modules**

The syntax of the type declarations follows the style that is used in the C programming language. A general template for a module is as follows, where the *implements* *ModuleCall* is optional:

```plaintext
module Name(decl₁,...,declₙ)
     implements ModuleCall {
         BodyStmt
     }
```

In the above template, Name is the name of the module and BodyStmt is the statement of the module that determines its behavior. The Name of a module may be any identifier that can also be used as a variable as explained in the previous chapter, but names of variables and modules have to be distinct. Each declᵢ is the declaration of one or more variables. These declarations have in turn the following syntax:

```
StorageClass type name₁ [n₁] ... [n₁ₙ₁],...,nameᵶ [n₁] ... [n₁ₙᵶ],
```

StorageClass is thereby either *event* or empty which means that the variable’s storage class is either *event* or memorized. The type *type* is specified next with the syntax we have already described in the previous chapter with that for an array type *array*(α, n) only the base type α is written at this place. The
remaining declaration consists of a comma-separated list of prefixed names of the declared variables. As already mentioned above, the names of controllable input variables and output variables have to be prefixed with the symbols \( ? \) and \( & \), respectively. For array types, the name of the variable is suffixed with the array dimensions embraced with square brackets.

In the frequent case \texttt{event bool} for the declaration of so-called pure signals, i.e. variables with storage class \texttt{event} of type \texttt{bool}, it is allowed to omit the type \texttt{bool}, so that \texttt{event} is an abbreviation of \texttt{event bool}. As an example, consider the following start of a module:

\[
\text{module M1(int<3> a[8], event \& o, nat<5> b, bool ?p, int \& r)}
\]

This declares a module with name \texttt{M1} having the arguments in its interface:

<table>
<thead>
<tr>
<th>number</th>
<th>name</th>
<th>type</th>
<th>storage class</th>
<th>dataflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>array(int&lt;3&gt;,8)</td>
<td>–</td>
<td>uncontrollable input</td>
</tr>
<tr>
<td>2</td>
<td>o</td>
<td>bool</td>
<td>event</td>
<td>output</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>nat&lt;5&gt;</td>
<td>–</td>
<td>uncontrollable input</td>
</tr>
<tr>
<td>4</td>
<td>p</td>
<td>bool</td>
<td>–</td>
<td>controllable input</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>int</td>
<td>memorized</td>
<td>output</td>
</tr>
</tbody>
</table>

\textit{Specification Modules}

The above explanations referred to behavioral modules that are used to model the system's behavior. In addition, Quartz also offers specification modules that do not model the behavior of a system, and are used instead to formally specify the properties that the module should fulfill. The syntax of specification modules is as follows:

\[
\text{spec Name(decl_1,...,decl_n) implements ModuleCall}\
\text{\quad name_1 : [task_1] spec_1;}\
\text{\quad \quad \quad \text{\vdots}}\
\text{\quad name_m : [task_m] spec_m;}
\]

Besides the use of the \texttt{spec} keyword instead of \texttt{module} there is only the difference that the body statement has been replaced with a nonempty list of specifications. This list contains entries of a name, a task and a specification. For the names, the rules we mentioned for variables in the previous chapter apply, and the specifications are also as explained in the previous chapter. It remains to discuss the optional task entry, which is as follows:

- \texttt{ProveA}: This task means that it has to be checked whether the property holds for \textit{all initial situations}.
- \texttt{ProveE}: This task means that it has to be checked whether the property holds for \textit{at least one initial situation}.
• **DisproveA**: This task means that it has to be checked whether the property is false for all initial situations.
• **DisproveE**: This task means that it has to be checked whether the property is false for at least one situation.

Note that the path quantifiers cannot be used to distinguish between the above cases. For example, ProveA: $A \phi$ means that it is to be checked whether in all initial states all paths satisfy the path property $\phi$, while ProveE: $E \phi$ means that it is to be checked whether there is an initial state such that all paths starting in that initial state satisfy the path property $\phi$. Analogously, ProveE: $A \phi$ means that it is to be checked whether there is a path satisfying $\phi$ in all initial states with a path starting in that state that satisfies $\phi$.

Note that all of the above tasks can be reduced to a standard model checking problem that checks whether all initial states are contained in the set of states that satisfy the given specification (note that $\mathcal{I} \neq \{\}$):

- **[ProveA]**: $\phi$ holds iff $\mathcal{I} \subseteq \llbracket \phi \rrbracket_K$.
- **[ProveE]**: $\phi$ holds iff $\mathcal{I} \nsubseteq \llbracket \neg \phi \rrbracket_K$, i.e. $\mathcal{I} \cap \llbracket \neg \phi \rrbracket_K \neq \{\}$.
- **[DisproveA]**: $\phi$ holds iff $\mathcal{I} \nsubseteq \llbracket \phi \rrbracket_K$.
- **[DisproveE]**: $\phi$ holds iff $\mathcal{I} \subseteq \llbracket \neg \phi \rrbracket_K$, i.e. $\mathcal{I} \cap \llbracket \neg \phi \rrbracket_K = \{\}$.

Even though the four tasks can be mapped to a standard model checking problem, it is necessary to distinguish at least two of them so that the model checker can report a success in all cases, while otherwise a failure would have to be reinterpreted as a success, which may lead to confusions.

In future versions of Quartz, more tasks will be available. In particular, tasks for supervisory control and worst case execution time analysis will be added.

While the meaning of a behavioral module is a state-based function that maps input values to output values, the meaning of a specification module is to check the tasks that are listed in the module. If the specification module is the first (or the only) module in a file, then this amounts to prove the properties for the universal Kripke structure that has one state for all possible variable assignments, and where all states are connected with each other. A [ProveA] task will then check the validity of the property, while a [ProveE] task checks whether the property is satisfiable. Analogously, [DisproveA] and [DisproveE] check whether the property is not valid and is not satisfiable, respectively.

If the specification module is not the first module in a file, then it is either dead code or it is called by the first module of that file. Specification modules can only be called by the implements clause in the header of a module. Different problems are specified depending on whether the specification module is called by a behavioral or a specification module, as is described in the next paragraph.
The Refinement Relation

Finally, the definition of modules also specifies the ‘implements’ relation between modules. This relation is a binary relation with the following meaning: module $M_1$ implements module $M_2$ iff the following holds:

- If both $M_1$ and $M_2$ are behavioral modules, then $M_1$ implements $M_2$ means that $M_1$ is a refined version of $M_2$ or in other words that $M_2$ is an abstraction of $M_1$. Formally, this means that there is a simulation relation between $M_1$ and $M_2$, so that $M_2$ can simulate each transition of $M_1$.
- If $M_1$ is a behavioral module and $M_2$ is a specification module, then $M_1$ implements $M_2$ means that all tasks listed in $M_2$ refer to the Kripke structure associated with $M_1$. Hence, the resulting problems lead to classic model checking problems.
- If both $M_1$ and $M_2$ are specification modules, then the conjunction of the properties of $M_1$ should imply the conjunction of the properties listed in $M_2$. The result is therefore a theorem proving task.
- The remaining case, where $M_2$ is a behavioral module and $M_1$ is a specification module, has not yet been given a meaning.

Simulation relations are very important to reduce the complexity of model checking problems. It is known that all universal properties (i.e., those without positive/negative occurrences of $E/A$) that have already been verified for $M_1$ do also hold for $M_2$. Thus, this can be used for compositional reasoning and abstract interpretation.

3.1.2 Statements

The semantics of a behavioral module mainly depends on the body statement that determines how the module will react to given input values. This requires to compute the output values for the current reaction step together with the internal state of the module for the next reaction step. The state of the module consists of the values of memorized outputs and the currently active locations in the statement. A statement is thereby called active if the control flow currently rest somewhere in that statement and will proceed with the execution from that location.

This leads to an important property of statements: There are statements that are executed in zero time, and others that require several reaction steps to be fully executed. Statements that are executed in zero time are called instantaneous. Instantaneous statements form so-called micro steps of the execution that are all executed in zero time. A finite number of micro steps is then collected in a macro step that consists of the code that is executed within one reaction. Note that the property of a statement to be instantaneous is a dynamic one since it depends on the current inputs and active control flow locations.
Table 3.1 lists all atomic statements that are available in Quartz. Statements that can hold the control flow are labeled with a location variable $\ell$ that is used to encode the control flow automaton of the statement. The control flow can rest at these locations so that we can view the location variables as boolean typed variables with the meaning that these variables are true iff the control flow currently rests at this location.

There are four action statements that are responsible for modifying the values of the output variables. All action statements are instantaneous, i.e., they are executed in zero time. Moreover, there are two kinds of assignment statements, namely the immediate assignment $x = \tau$; that modifies the value of the variable $x$ instantaneously so that $x$ has the same value as $\tau$. Hence, the equation $x == \tau$ holds in the reaction step where the assignment is executed. The delayed assignment $\text{next}(x) = \tau$; is also executed in zero time as follows: in the current reaction step, the expression $\tau$ is evaluated to a value that will be the value of the variable $x$ in the next reaction step.

The expression $\tau$ is thereby an arbitrary expression that must have the same type as the left hand side of the assignment. We have already discussed in Section 2.1.2 that it is tolerated by the language that the left hand side may have too few bits to store the entire value of the right hand side, but except for different ranges, the types must be the same.

The left hand side $x$ must be one of the following:

- a single variable of atomic type
- an array access
- a bitvector slice/bit
- a tuple component access
Therefore, assignments can be used to change all parts of a variable of an arbitrary type. The remaining action statements `emit x;` and `emit next(x);` are syntactic sugar and are retained for historical reasons: `emit x;` abbreviates `x = true;` and `emit next(x);` abbreviates `next(x) = true;`, where it is moreover required that `x` is a boolean variable with storage class `event`.

Assumptions and assertions do neither modify output variables nor do they influence the control flow. The statement that would be obtained by removing assumptions and assertions will be equivalent. An assumption `assume(\sigma);` with a boolean expression `\sigma` is executed in zero time and simply tells the compiler that it can believe that the condition `\sigma` holds in this macro step. In contrast to this, an assertion `assert(\sigma);` instructs the compiler to check whether the boolean condition `\sigma` holds in the current macro step. The compiler collects all assertions and assumptions and generates corresponding verification problems.

Some previous versions of Quartz moreover contained the requirement statement `now(\sigma)`. Similar to `assume(\sigma)`, also the statement `now(\sigma)` states that the programmer states here that the condition `\sigma` holds. However, `now(\sigma)` eliminates all states (and related transitions) that violate that condition, while `assume(\sigma)` retains these unwanted states. In Quartz 2.0, this distinction is no longer made, and instead, a different use of assumptions is proposed in code synthesis or model checking: Code synthesis may generate observers, and model checking can remove the unwanted states as it was formerly the case with the `now` statement.

The remaining atomic statements concern the propagation of the control flow. `nothing` is the empty statement, which does neither modify the program's variables nor does it stop the control flow (hence, it is instantaneous just like the actions, assumptions and assertions). The `pause` statement is very important: it denotes the end of the current macro step and the beginning of the next macro step. If `\ell:pause` is executed in the current macro step, then the execution of the current macro step is completed and the execution will proceed in the next macro step from location `\ell`. The statement `\ell:halt` stops the program's execution by entering an infinite loop of `pause` statements, so that we define the semantics of `\ell:halt` as `loop \ell:pause`.

The `await` statements depend on a boolean condition `\sigma`. As the name suggests, the statement intuitively waits at the `await` statement until the boolean condition `\sigma` holds. In more detail, consider first the immediate variant: if `\ell:immediate await(\sigma);` is executed, it is checked whether `\sigma` holds in the current macro step. If `\sigma` holds, then the execution can proceed with further micro steps, so that the statement behaves as `nothing`. Otherwise, if `\sigma` does not hold, the execution is stopped and the control flow is caught at the `await` statement for the next macro step. At the next macro step, the execution starts then again by executing `\ell:immediate await(\sigma);` so that the control flow waits at this location until `\sigma` holds.

The delayed variant `\ell:await (\sigma);` differs as follows: if `\ell:await (\sigma);` is executed in the current macro step, the control flow is stopped independent
of the value of \( \sigma \). That is, it is not checked whether \( \sigma \) holds or not, instead, \( \ell : \text{await}(\sigma) \) behaves like \( \ell : \text{pause} \) when started. In the next macro step, the immediate wait statement \( \ell : \text{immediate wait}(\sigma) \) is then executed, so that the control flow waits in location \( \ell \) until the condition \( \sigma \) holds. Below, we will define \( \ell : \text{await}(\sigma) \) as a macro for \( \text{do} \ell : \text{pause}; \text{while}(!\sigma) \) and analogously, \( \ell : \text{immediate await}(\sigma) \) as a macro for \( \text{while}(!\sigma) \ell : \text{pause} ; \), which succinctly explains the differences.

Table 3.2 shows the compound statements of Quartz, which are statements that are constructed by already existing statements. The first group of statements are conditional statements that execute one or the other substatement. The conditional (if-else) statement is typical for every imperative programming language. Depending on the value of \( \sigma \) in the current macrostep, the conditional statement \( \text{if}(\sigma) S_1 \text{ else } S_2 \) either \( S_1 \) or \( S_2 \) is immediately executed. That is, the condition \( \sigma \) is evaluated in zero time with the variable assignment of the current macro step, and if the result is true, then the statement \( S_1 \) is immediately started, otherwise \( S_2 \) is immediately started. Hence, depending on the evaluation of \( \sigma \), \( \text{if}(\sigma) S_1 \text{ else } S_2 \) behaves either as \( S_1 \) or \( S_2 \). The variant \( \text{if}(\sigma) S_1 \) is an abbreviation for \( \text{if}(\sigma) S_1 \text{ else nothing} ; \).

The \text{choose} statement chooses nondeterministically at runtime whether \( S_1 \) or \( S_2 \) is executed. Hence, it behaves either like \( S_1 \) or \( S_2 \). Note that hardware and software synthesis is not directly possible for nondeterministic statements like the \text{choose} statement as will be discussed in detail in Section 3.4.10.

The \text{switch} statement is a multiple case selection. The meaning is that the boolean conditions \( \sigma_i \) of the case clauses in the statement are checked one after the other. The first substatement \( S_i \) whose boolean condition \( \sigma_i \) holds is then executed. If none of the boolean conditions hold, then the mandatory statement after the \text{else} clause is executed.

The next group of statements is concerned with the sequential and parallel control flow: Writing statements in sequence \( S_1 S_2 \) is interpreted as sequential execution which is as follows: \( S_1 \) is immediately started when the sequence \( S_1 S_2 \) is started. Then, we have to distinguish two cases:

- If the execution of \( S_1 \) is instantaneous, then also \( S_2 \) is started in the same macro step.
- If the execution of \( S_1 \) takes time, then the control flow will stop at some locations inside \( S_1 \) and will proceed with the execution from those locations at the next macro step. It can be the case that \( S_1 \) will never terminate so that \( S_2 \) is never started. Otherwise, \( S_2 \) is started in the macro step where \( S_1 \) terminates.

Synchronous languages like Quartz are concurrent languages which is first due to the concurrent execution of the micro steps in one macro step, and second due to the availability of statements for the execution of parallel threads. In Quartz, we have three parallel statements: \( S_1 || S_2 \) denotes the synchronous parallel execution of \( S_1 \) and \( S_2 \). If \( S_1 || S_2 \) is started, we start both \( S_1 \) and \( S_2 \), and proceed with the concurrent execution of both statements. If the execu-
### Conditional Statements
- `if(σ) S_i` (conditional statement)
- `if(σ) S_i else S_j` (conditional statement)
- `choose S_i else S_j` (nondeterministic choice)
- `switch
  case(σ_1) do S_i
  ...
  case(σ_n) do S_n
else S` (case statement)

### Sequential and Parallel Control Flow
- `S_1 S_2` (sequential execution)
- `S_1 || S_2 and S_1 && S_2` (synchronous parallel execution)
- `S_1 || S_2 and S_1 &&& S_2` (asynchronous parallel execution)
- `S_1 || S_2 and S_1 &&& S_2` (interleaved parallel execution)

### Generic Sequential and Parallel Control Flow
- `sequence(α x=τ..π where σ) S` (generic sequence)
- `parallel(α x=τ..π where σ) S` (generic synchronous parallel statement)
- `async(α x=τ..π where σ) S` (generic asynchronous parallel statement)
- `interleave(α x=τ..π where σ) S` (generic interleaved parallel statement)

### Loops
- `do S while(σ);` (do-loop)
- `while(σ) S` (while-loop)
- `loop S` (infinite loop)
- `ℓ:loop S each(σ);` (triggered infinite loop)
- `ℓ:every(σ) S` (triggered infinite loop)

### Abortion
- `weak immediate abort S when(σ);` (weak immediate abortion)
- `weak abort S when(σ);` (weak abortion)
- `immediate abort S when(σ);` (strong immediate abortion)
- `abort S when(σ);` (strong abortion)

### Suspension
- `ℓ:weak immediate suspend S when(σ);` (weak immediate suspension)
- `weak suspend S when(σ);` (weak suspension)
- `ℓ:immediate suspend S when(σ);` (strong immediate suspension)
- `suspend S when(σ);` (strong suspension)

### Exceptions
- `throw e;` (throw exception e)
- `try(e) S catch(e) S_e` (declare and catch exception e)

### Miscellaneous
- `{α x_1,...,x_n; S}` (local declaration)
- `let(α x=τ) S` (let-abbreviation)
- `during S_1 do S_2` (during statement)
- `C:name(τ_1,...,τ_n); S` (module instantiation)
- `{S}` (statement block)

### Table 3.2. Compound Statements in Quartz
tion of both $S_1$ and $S_2$ is instantaneous, then also $S_1 \parallel S_2$ instantaneously terminates and further micro steps can be executed in the current macro step. Otherwise, the execution of $S_1 \parallel S_2$ takes time. As long as the control flow rests at locations somewhere inside both $S_1$ and $S_2$, both threads are executed in lockstep. If $S_1$ terminates, but $S_2$ does not, then $S_1 \parallel S_2$ behaves further as $S_2$ does (and vice versa). If finally $S_2$ terminates, then $S_1 \parallel S_2$ also terminates.

Note that the synchronous parallel execution $S_1 \parallel S_2$ is a deterministic statement which is remarkable for concurrent languages. In contrast, the asynchronous parallel execution $S_1 \| S_2$ behaves at starting time exactly as $S_1 \parallel S_2$, and if at least one of $S_1$ or $S_2$ is instantaneous, then there is still no difference. However, if both threads $S_1$ and $S_2$ are active, then it is possible that the control flow executes both threads in lockstep (as in case of $S_1 \parallel S_2$), or that only $S_1$ is executed, but $S_2$ is not further executed, or vice versa that $S_2$ is executed and $S_1$ is suspended. Hence, the statement is nondeterministic in this case.

The interleaving statement $S_1 \mid S_2$ is also nondeterministic when both threads hold the control flow. However, instead of having the choice to execute either one of $S_1$ or $S_2$ or both, the interleaving statement $S_1 \mid S_2$ does not have the freedom to execute both threads. Instead, it executes exactly one of the active threads $S_1$ and $S_2$ as long as both are active.

The conjunctively active variants $S_1 \& S_2$, $S_1 \&\& S_2$, and $S_1 \& S_2$ differ in that these statements terminate as soon as the first thread $S_i$ terminates (while $S_1 \parallel S_2$, $S_1 \| S_2$, and $S_1 \mid S_2$ terminate when the last of the two threads terminates). Therefore, $S_1 \& S_2$, $S_1 \&\& S_2$, and $S_1 \& S_2$ are called conjunctively active statements, while $S_1 \parallel S_2$, $S_1 \| S_2$, and $S_1 \mid S_2$ are called disjunctively active parallel statements. As long as both threads are active, there is no difference between these variants.

For the sequential and the parallel execution statements, there are also generic versions that are listed in the next group of statements in Table 3.2. These statements instantiate their body statement $S$ with those values $\tau, \ldots, \pi$ that satisfy the boolean condition $\sigma$, and the obtained statements are combined with the corresponding sequential or parallel operator. The where clause with the boolean filter condition $\sigma$ is optional. Note that the sequence operator as well as all parallel statements are associative, so that for the definition of the semantics, it does not matter how the associativity is defined. Nevertheless, for parsing a definition is necessary as given below.

The next group of statements are different kinds of loops. The do $S$ while($\sigma$); loop starts $S$ at starting time. The execution of $S$ must not be instantaneous, since otherwise a macro step with infinitely many micro steps would be generated. Of course, $S$ may never terminate. However, if $S$ terminates at some point of time, then it is checked in that macro step whether $\sigma$ holds. If this is the case, then the loop is restarted, otherwise the loop also terminates.
The \texttt{while}(\sigma) S \texttt{statement} differs in that the loop condition \( \sigma \) is checked at starting time and the statement instantaneously terminates if the result is false. Otherwise the semantics is the same as \texttt{do} S \texttt{while}(\sigma);. Hence, we can define \texttt{while}(\sigma) S as \texttt{if}(\sigma) \texttt{do} S \texttt{while}(\sigma);. The infinite loop \texttt{loop} S can be defined as \texttt{while(true)} S, so that it infinitely often iterates the statement S.

The triggered loop \( \ell:\texttt{loop} S \texttt{each}(\sigma); \) starts S if the loop is started. If during the execution of S the condition \( \sigma \) holds, then S is aborted and restarted. Otherwise, it may be the case that S will never terminate, or that it finally will terminate. If S terminates, then the statement waits until \( \sigma \) holds, and as soon as this is the case, the entire statement is restarted.

The every statement \((\ell_1,\ell_2):\texttt{every}(\sigma) S \texttt{differs from} \ell:\texttt{loop} S \texttt{each}(\sigma)\) in that at starting time, the statement first waits on \( \sigma \), i.e., we can define the every statement as an abbreviation of \( \ell_1:\texttt{await}(\sigma); \ell_2:\texttt{loop} S \texttt{each}(\sigma); \). There are also immediate variants \( \ell:\texttt{immediate loop} S \texttt{each}(\sigma); \) and the corresponding \((\ell_1,\ell_2):\texttt{immediate every}(\sigma) S \).

The next two blocks are different variants of suspension and abortion statements: \texttt{abort} S when(\sigma) immediately enters S at starting time (regardless of the current value of \( \sigma \)). If the execution of S is instantaneous, so is the execution of \texttt{abort} S when(\sigma). Otherwise, the control flow is caught somewhere inside S for the execution in the next macro step. However, if in the next macro step, the condition \( \sigma \) holds, then the execution of S is aborted and \texttt{abort} S when(\sigma) terminates. Hence, S is started and then executed as long as \( \sigma \) is 0.

The immediate variant \texttt{immediate abort} S when(\sigma) differs in that the condition \( \sigma \) is already evaluated at starting time. If \( \sigma \) holds at starting time, then the entire statement behaves like nothing, otherwise it is the same as \texttt{abort} S when(\sigma). Hence, we can define the immediate variant as \texttt{if}(\sigma) \texttt{nothing}; else \texttt{abort} S when(\sigma).

The ‘weak’ variants of abortion differ on the data actions at abortion time: While the strong variants ignore all data manipulations at abortion time, all of them are performed by the weak variants. Note that this also holds for \texttt{assume} and \texttt{assert} statements. Note further that the control flow of the weak and strong variants is the same; only the data flow at abortion time is different.

Similar to \texttt{abort} S when(\sigma), the statement \texttt{suspend} S when(\sigma) immediately enters S at starting time (regardless of the current value of \( \sigma \)). If the execution of S is instantaneous, so is the execution of \texttt{suspend} S when(\sigma). Otherwise, the control flow is caught somewhere inside S for the execution in the next macro step. However, if in the next macro step, the condition \( \sigma \) holds, then the execution of S is suspended and the execution proceeds at the next macro step from the locations that were active at the previous macro step. Hence, S is suspended whenever \( \sigma \) holds, except for the macro step where the statement is started.

Analogous to the immediate abortion, the immediate variant of the suspension statement \texttt{immediate suspend} S when(\sigma) differs in that the con-
dition \( \sigma \) is already evaluated at starting time. If \( \sigma \) holds at starting time, then the entire statement behaves like \( \ell: \text{pause} \), otherwise it is the same as \( \text{suspend } S \text{ when}(\sigma) \). Note that the immediate form introduces another control flow location that does not exist in the delayed version.

The weak variants of suspension differ in those macro steps where the suspension takes place: Whenever \( \sigma \) holds, the data actions are all executed in the weak version, while none of them are executed in the strong suspension statements.

Suspension and abortion statements are often called \textit{preemption statements}, since the control flow of their body \( S \) can be preempted when the preemption condition \( \sigma \) holds. The exception handling statements are discussed in detail in Section 3.4.11.

The statement \{\( \alpha \ x_1, \ldots, x_n ; S \)} declares local variables \( x_1, \ldots, x_n \). The type and storage class (\textit{event} or memorized) is given by the declaration \( \alpha \), and the scope is the body statement \( S \). The meaning of the statement is the meaning of \( S \), however, with the additional knowledge of the local variables. The local variables \( x_1, \ldots, x_n \) are not seen outside the local declaration, and are shadowed if already variables of this name exist outside the scope. However, shadowing is a bad feature, so that it is recommended for compilers to forbid this feature.

The abbreviation statement \texttt{let(x=\tau) S} has the meaning that all occurrences of the variable \( x \) in \( S \) can be replaced by the expression \( \tau \). Hence, \( x \) abbreviates \( \tau \) in \( S \) and therefore allows one to abbreviate large expressions by variables with hopefully self-explanatory names. If common expressions are shared, there is no reason to not replace the occurrences of \( x \) by \( \tau \) before compilation.

The statement \texttt{during } S_1 \texttt{ do } S_2 \texttt{ starts } S_1 \texttt{ at starting time. If the execution of } S_1 \texttt{ is instantaneous, then } \texttt{during } S_1 \texttt{ do } S_2 \texttt{ is the same as } S_1 \texttt{. Otherwise, in each macro step of } S_1 \texttt{ after starting time the instantaneous statement } S_2 \texttt{ is executed except for the termination time of } S_1 \texttt{. For example, the during statement can be used to express invariants in that } S_2 \texttt{ is an assertion statement. As already mentioned, } S_2 \texttt{ must be instantaneous, and this must be statically checkable, which means that } S_2 \texttt{ must not contain any statement that contains location variable.}

Quartz modules can be instantiated in statements by the module call \( C : \text{name}(\tau_1, \ldots, \tau_n) \). This means that the body statement of module \texttt{name} is executed with the replacements of the expressions \( \tau_i \) for the input and output variables that were used in the declaration of module \texttt{name} with the following restrictions:

- Arbitrary expressions can be used to replace input variables as long as the corresponding types match (except for a different range as explained in Section 2.1.2).
- Output and local variables of the current module can be used to replace the output variables of the called module as long as the types are the same.
3.1 Modules and Interface Declarations

Note that module definitions and module instantiation are comparable to class definitions and object instantiations as used in many object oriented languages. In particular, one and the same module can be instantiated many times and different expressions can be used for these instantiations. The optional name \( C \) can be used to distinguish the different instantiations.

<table>
<thead>
<tr>
<th>priority</th>
<th>statement operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unary operators: ( \text{if(}\sigma\text{)}, \text{while(}\sigma\text{)}, \text{loop} )</td>
</tr>
<tr>
<td>2</td>
<td>( &amp;&amp; )</td>
</tr>
<tr>
<td>3</td>
<td>( &amp;&amp;&amp; )</td>
</tr>
<tr>
<td>4</td>
<td>( &amp; )</td>
</tr>
<tr>
<td>5</td>
<td>(</td>
</tr>
<tr>
<td>6</td>
<td>( ||| )</td>
</tr>
<tr>
<td>7</td>
<td>(</td>
</tr>
<tr>
<td>8</td>
<td>sequences</td>
</tr>
<tr>
<td>9</td>
<td>local declarations</td>
</tr>
<tr>
<td>10</td>
<td>( \text{else}, \text{each} )</td>
</tr>
</tbody>
</table>

Table 3.3. Precedences of Statement Operators

Finally, the statement block \( \{S\} \) is used to break the precedences of the infix statement operators that are listed in Table 3.3. Unary operators are thereby all statement operators (possibly containing an expression) that are applied to a single statement. In particular, \( \text{if(}\sigma\text{)}, \text{while(}\sigma\text{)}, \text{loop} \), and generic sequences and concurrency statements are unary statement operators. The infix operators \( \&\& \), \( \&\&\& \), \( \& \), \( || \), \( \|\|\| \), \( | \), and the sequence operator have the priorities as listed in Table StatementPrecedencesTable. There is no need to discuss priorities of the other statement operators, since these have textual start and end parts so that the operand statements are unambiguously determined.

As an example, \( S_1; S_2 \| S_3 \) is parsed as \( S_1; \{S_2 \| S_3\} \), so that delimiters are required for the statement \( \{S_1; S_2\} \| S_3 \).

Moreover, the language suffers from the ‘dangling else’ and ‘dangling each’ problem for conditionals and \( \text{loop} \ldots \text{each} \) statements, which are solved as usual: an \text{else} branch refers to the first upper if statement such that there is neither an \text{else}-branch in between. Analogously, an \text{each} part refers to the first preceding \text{loop} \( S \) that is not closed by another \text{each} part. In case these rules should be in conflict, the innermost one has precedence.

The concrete syntax is given in Appendix C.1 that contains the precise specification of a lexer and parser for the Quartz language.
3.2 Semantic Problems

Synchronous languages like Quartz suffer from special problems that have to be handled by the compilers. In this section, we informally present these problems to introduce the notions of schizophrenia and causality that are considered in more detail in the following chapters.

3.2.1 Schizophrenic Statements

Schizophrenic statements are statements that have several active instances at the same macro step. This may happen only if the statement is contained in a loop, since it is required to restart the statement at termination or abortion time. If the statement is contained in $n$ nested loops, it may even be the case that $n$ instances exist at the same point of time, for example, if all of the loops are restarted and aborted at the same point of time.

As variables have a uniquely determined value for each macro step, it is seems not to be problematic if one and the same statement is started several times in the same macro step. However, the scopes of the local variables are entered and left in micro steps of the execution, so that the different instances of the schizophrenic statement may have to consider different scopes of local variables. This leads to the notion of so called reincarnations of local variables.

module Schizophrenic(event &x0,&x1,&x2,&x3) {
  loop {
    event x;
    if(x) emit x1; else emit x0;
    assert(!x);
    pause;
    emit x;
    if(x) emit x3; else emit x2;
    assert(x);
  }
}

Fig. 3.1. Schizophrenic local declaration

As an example, consider the module given in Figure 3.1. This module has only one control flow location which is the pause statement in its loop body. If the execution proceeds from that location, it emits the local variable $x$, so that the following conditional statement emits $x_3$. Clearly, the assertion $\text{assert}(x)$ holds. However, as the loop body terminates, also the scope of the local declaration is left, and the loop body is restarted due to a new loop iteration. For this reason, a new scope of the local declaration statement is opened, and a new incarnation of the local variable $x$ is generated. Hence, there are now two instances of the local variable $x$: one living in the scope
of the previous loop body, and another one living in the scope of the new loop body. While the older incarnation is true, the new one is false, so that the incarnations have different values. Hence, also the assertion \( \text{assert}(\neg x) \) holds, which is not a contradiction, since the two assertions refer to different incarnations of \( x \).

A program with multiple schizophrenic instances is shown in Figure 3.2. When started, the control flow occupies the three locations \( \text{ell0} \), \( \text{ell1} \), \( \text{ell2} \), and \( \text{ell3} \) and no signals are emitted except for \( y_{000} \). At the next point of time, a lot of micro steps are executed:

- The local variables \( x_1 \), \( x_2 \), and \( x_3 \) are emitted by the three threads that start from locations \( \text{ell1} \), \( \text{ell2} \), and \( \text{ell3} \), respectively. The innermost loop iterates and executes the \text{switch} statement that therefore emits \( y_{111} \). This thread aims at stopping at location \( \text{ell0} \), but is aborted as will be explained next.
- Since \( x_3 \) holds, the \text{weak abort} statement with abortion condition \( x_3 \) aborts its body statement. As this \text{weak abort} statement is contained in a loop, the loop body is restarted, so that a new scope of \( x_3 \) is generated. This thread then stops at locations \( \text{ell3} \) and \( \text{ell0} \) after having again executed the \text{switch} statement that emits this time \( y_{110} \).
- Concurrently, the \text{weak abort} statement with abortion condition \( x_2 \) aborts its body statement, and the surrounding loop body is restarted, so that new scopes of \( x_2 \) and \( x_3 \) are generated. This thread then stops at locations \( \text{ell2} \), \( \text{ell3} \) and \( \text{ell0} \) after having once more executed the \text{switch} statement that emits this time \( y_{100} \).
- Finally, the \text{weak abort} statement with abortion condition \( x_1 \) also aborts its body statement, and the surrounding loop body is restarted, so that further new scopes of \( x_1 \), \( x_2 \) and \( x_3 \) are generated. This thread then stops at locations \( \text{ell1} \), \( \text{ell2} \), \( \text{ell3} \) and \( \text{ell0} \) after having once more executed the \text{switch} statement that emits this time \( y_{000} \).

Hence, after the starting point of time, the signals \( y_{000} \), \( y_{100} \), \( y_{110} \), and \( y_{111} \) are emitted, while the other outputs are never emitted. The local variables \( x_3 \), \( x_2 \), and \( x_1 \) have three, two and one incarnations that concurrently exists with the incarnation of the old scope.

Schizophrenic statements are not a big problem. The semantics and the compilation techniques that are described in the following chapters all cope with this issue. In particular, schizophrenia problems can be easily solved by generating copies of internal data structures that are used for compilation like hardware gates or nodes in a data flow graph. Moreover, the blow-up that occurs due to these necessary copies is polynomial: We will see in the later chapters that a quadratic blow-up suffices. This quadratic blow-up is obtained only with extreme examples like the one of Figure 3.2, and does seldom occur in practice.

Hence, schizophrenia problems are neither a problem for state-of-the-art compilers nor for the definition of the language’s semantics. However, pro-
module Gonthier02(event &y111,&y110,&y101,&y100,
     &y011,&y010,&y001,&y000) {

loop {
    event x1;
    weak abort
    {
        ell1:pause;
        emit x1;
    }
    ||
    loop {
        event x2;
        weak abort
        {
            ell2:pause;
            emit x2;
        }
        ||
        loop {
            event x3;
            weak abort
            {
                ell3:pause;
                emit x3;
            }
            ||
            loop {
                switch
                (!x1 & !x2 & !x3) do emit y000;
                (!x1 & !x2 & x3) do emit y001;
                (!x1 & x2 & !x3) do emit y010;
                (!x1 & x2 & x3) do emit y011;
                ( x1 & !x2 & !x3) do emit y100;
                ( x1 & !x2 & x3) do emit y101;
                ( x1 & x2 & !x3) do emit y110;
                ( x1 & x2 & x3) do emit y111;
                else nothing;
                ell0:pause;
            }
        when(x3);
    }
        when(x2);
    }
    when(x1);
}

Fig. 3.2. Multiply schizophrenic local declaration
grammers must be aware of schizophrenia problems that might sometimes be confusing.

### 3.2.2 Causality

Besides schizophrenic statements, causality problems are a major burden for compilers of synchronous languages. Intuitively, causality cycles are obtained when an action instantaneously changes its precondition by its execution. Since the effect of the action must be instantaneously observed, we may end up with a contradiction to its precondition. Compilers for synchronous languages must be able to analyse the causality of programs to avoid that the generated systems suffer from runtime problems like deadlocks.

There are different equivalent interpretation of causality in terms of constructive logic, stability of asynchronous circuits with combinational cycles, and the existence of dynamic schedules as we will discuss in later chapters [42]. Moreover, the definition of causally correct programs depends on the definition of the semantics of the programs that can be changed to some extent [53, 226, 229, 230]. Of course, only modifications of the definition of causality make sense that have equivalences in constructive logic, in asynchronous circuits, and dynamic schedules. We will discuss possible choices of the semantics in the next chapter, the relationship to hardware circuits in the chapter on hardware synthesis, and the relationship to dynamic schedules in the software synthesis chapter.

Causality problems are best explained by some simple examples. To this end, we use example programs given in the literature like in particular in [42] and [53]. These programs do only make use of boolean typed variables of event storage mode, but, of course, causality is also an issue for other outputs. To start with, consider the programs given in Figure 3.3.

Module P01 has no particular problem. Note however, that even though all of the three conditional statements are instantaneous, and are therefore executed at the same time, it is necessary to read (and execute) the statements in the given order: If input $i$ is present, then we emit $o_1$, and therefore we do neither emit $o_2$ nor $o_3$. Otherwise, if input $i$ is absent, then we do not emit $o_1$, and therefore we do emit both $o_2$ and $o_3$. Hence, even though the micro steps are executed in the same macro step, there is still an order defined on them that is determined by their data dependencies.

The data dependencies of module P01 follow the order of the conditional statements as listed in the program. If the conditional statements in the sequence would be rearranged, we still would have the same unique behavior from a logical point of view. However, it would no longer be the case that we could execute the program from top to bottom. Instead, we would still have to execute the conditional statements in the order given in module P01. We will later see that the rearranged conditional statements may not be causally correct (this depends on the definition of causality), while the order given in P01 is causally correct. Finally, we remark that using a parallel statement would
module P01(event i,&o1,&o2,&o3) {
    if(i) emit o1;
    if(!o1) emit o2;
    if(o2) emit o3;
}

module P02(event &o1,&o2) {
    emit o2;
    if(o1) {
        if(o2) pause;
        emit o1;
    }
}

module P03(event &o) {
    if(!o) emit o;
}

module P04(event &o) {
    if(o) emit o;
}

module P05(event &o1,&o2) {
    if(o1) emit o2;
    if(!o2) emit o1;
}

module P06(event &o1,&o2) {
    if(o1) emit o2;
    if(o2) emit o1;
}

Fig. 3.3. Simple Causality Problems (Part I of III)

allow us to use any ordering of the conditional statements. For this reason, sequences and parallel statements are even different if the used substatements are instantaneous.

Module P02 definitely first emits o2. However, the conditional statement then asks whether o1 holds, which is unclear at that stage, and for this reason, module P02 may be viewed as causally incorrect. Looking forward into the branches of the conditional statement, we see however that o1 can not be emitted in the first macro step, since the presence of o2 catches the control flow at the pause statement. Therefore o1 is not present so that the condition statement executes its empty else branch. Module P02 is therefore instantaneous and behaves like emit o2.

Module P03 is the smallest module with a causality conflict that leads to a paradoxon: When the conditional statement is executed, it is unclear whether
3.2 Semantic Problems

If o holds or not. However, if we make a case distinction and first speculate that o may be false, we would execute emit o, and therefore o would be true. In the remaining case, where we assume that o is true, we would not execute emit o, and as there is no emission of o and o has an event storage mode, o would be false. Hence, module P03 does not have a logically consistent behavior which must be forbidden to achieve deterministic programs.

In contrast to module P03, module P04 has two logically consistent behaviors: If we assume that o is false, then we do not execute emit o, which is consistent with our assumption. Otherwise, if we assume that o is true, then we do execute emit o, which is also consistent with our assumption. Hence, module P04 has two consistent behaviors, which must be forbidden to achieve deterministic programs.

Modules P05 and P06 are similar to modules P03 and P04, respectively: Module P05 has no logically consistent behavior, while module P06 has the two logical consistent behaviors that satisfy the formula o1 <-> o2.

Module P07 does not have a consistent logical behavior: If we assume that o would be true, we would have to stop at the pause statement, and therefore we would not emit o, so that the assumption that o is true would be wrong. If we instead assume that o would be false, we would execute emit o which makes o true. Hence, P07 does not have a consistent logical behavior.

Module P08 is a bit more complex and shows that programs can be logically correct only for some inputs: Consider first the case where i is false. Hence, the first thread of the body of the abort statement stops at the pause statement, so that emit o1 is not executed. The second thread asks whether o1 holds or not, which is unclear at that stage, so that we have to distinguish two cases:

- Assume first that o1 would be false. Then, we do not execute emit o2, and therefore both o1 and o2 are false. The second thread terminates, and the first one resumes its execution in the next macro step by executing emit o1, which is once more executed after the abortion statement.
- Now assume that o1 would be true. Then, we execute emit o2 in the second thread, and therefore both o1 and o2 are true. Since o2 is present, the abortion takes place, so that the control flow does not stop at the pause statement, and instead the emit o1 statement after the abortion is executed which makes our assumption consistent.

Hence, if i is false, we have two logically consistent behaviors that satisfy the formula o1<->o2. As we are interested in deterministic programs, this must be forbidden by the semantics of the language (as described in the next chapter). It remains to consider the case where i is true. In this case, the first thread executes emit o1, so that also o1 is true. Therefore, the second thread emits o2, so that the parallel statement terminates. At the same time it is also aborted, but since the abortion is weak, the emissions remain valid. The further emission of o1 does not harm the semantics in this case. Hence, module P08 has a causally correct behavior iff i is true.
module P07(event &o) {
    if(o) pause;
    emit o;
}

module P08(event i,&o1,&o2) {
    weak immediate abort {
        {
            if(!i) pause;
            emit o1;
        }
        ||
        if(o1) emit o2;
    } when(o2);
    emit o1;
}

module P09(event &o1,&o2) {
    if(o1) emit o1;
    ||
    if(o1)
        if(o2) nothing;
        else emit o2;
}

module P10(event &o) {
    if(o) nothing;
    emit o;
}

module P11(event &o1,&o2) {
    if(o1) {
        emit o2;
        if(o2) pause;
        emit o1;
    }
}

module P12(event &o) {
    if(o)
        emit o;
    else
        emit o;
}

Fig. 3.4. Simple Causality Problems (Part II of III)
3.2 Semantic Problems

Module P09 has a unique logically consistent behavior: If o1 is false, nothing is emitted, so that both o1 and o2 are false. If we would assume that o1 is true, then we would have to execute the nested conditional statement, which leads to a contradiction as explained in the discussion of module P03. Module P09 is therefore interesting since it asks whether we want to allow a module to hide incorrect statements.

Module P10 has clearly the unique logical behavior that is equivalent to emit o. This module is one of the modules to show different notions of causality. There is a strict notion of causality that would argue that P10 is not correct, since the conditional statement asks for o at a situation where we have not yet seen an emission of that signal. However, with a bit more intelligence, the compiler will see that the conditional statement is in all cases instantaneous, and therefore emit o is executed independent on the execution of the then- or else-branch of the conditional statement.

The unique possible behavior of module P11 is that both o1 and o2 are false, so that the module is equivalent to nothing. If o1 would be true, we would also emit o2, and therefore the control flow would be stopped at the pause statement. However, then we do not have an emission of o1, so that o1 must be false, which contradicts our assumption. P11 is again a program that has a unique behavior that is however only seen by speculation, which is extremely expensive (exponential) for many variables.

Module P12 is again a module that is used to discuss different notions of causality. A strict definition would forbid the program, since it also has to speculate when executed in the strict program order. However, a simple logical simplification shows that o is emitted independent on whether the then- or else-branch of the conditional statement is executed, and therefore the unique behavior of P12 would be viewed as causally correct.

Modules P13 up to P19 will be discussed later on in more detail, since they also discuss subtle differences on the notion of causality. We therefore only briefly list what their logical behavior is: Module P13 has the single unique behavior that both o1 and o2 must be false, since the assumption that one of these signals would be true can not be justified since the program can not execute a emission for that output. For similar reasons, P14 has the single unique behavior that both o1 and o2 must be false.

Module P15 clearly first emits o2. However, then we have to speculate whether o1 holds or not. Since, we already known that o2 is true, we can see that emit o1 can never be executed. Whether speculation on the value of o1 is necessary or not depends on subtle definitions of the semantics as we will discuss in the following chapter.

The logical behavior of module P16 is clear: o must be false, since the condition for executing emit o is o&!o which is not satisfiable. However, checking the satisfiability of formulas is expensive, and therefore we have to find other arguments to determine the behavior of P16 (as will be explained in the next chapter).
module P13(event i,&o1,&o2) {
    if(i)
        if(o1) emit o2;
    else
        if(o2) emit o1;
}

module P14(event &o1,&o2) {
    if(o1) emit o2;
    pause;
    if(o2) emit o1;
}

module P15(event &o1,&o2) {
    emit o2;
    if(o1)
        if(!o2) emit o1;
}

module P16(event &o) {
    if(o)
        if(!o) emit o;
}

module P17(event &o1,&o2) {
    if(o1) {
        emit o2;
        if(!o2) emit o1;
    }
}

module P18(event &o1,&o2) {
    if(o1) {
        emit o2;
        ||
        if(!o2) emit o1;
    }
}

module P19(event &o1,&o2) {
    if(o1) {
        emit o2;
        ||
        if(o2) emit o1;
    }
}

Fig. 3.5. Simple Causality Problems (Part III of III)
3.3 Core Statements

The situation is similar for module P17: The unique behavior is that neither o1 nor o2 is emitted, since the preconditions to execute emit o1 and emit o2 are o1&!o2 and o1, respectively, which can not be satisfied by other variable assignments. The same arguments (and even the same preconditions) apply for module P18. Then changed condition of the second conditional which leads to module P19 allows two behaviors that satisfy o1 <-> o2.

As can be seen by the above examples, the analysis of the behavior of synchronous programs can become rather complex due to the instantaneous reaction. The semantics of the language has to incorporate some means to check the causality of programs in an efficient way. We will define this in detail in the next chapter where the formal definition of the semantics is presented. There is some freedom for the definition of causally correct programs that also determines whether certain logical simplifications are valid. In general, causal correctness clearly depends on the syntax of the program, so that most programs can be easily rewritten to become causally correct.

3.3 Core Statements

In the previous section, we have discussed all the available statements of the Quartz language. With minor exceptions, these statements are similar to the statements that have been defined in the Esterel language. Many of the statements can be expressed in terms of other statements, so that a lot of syntactic sugar is contained in the set of statements. The syntactic sugar is important to express directly what is meant by the programmer, but it makes the following explanations of the formal semantics as well as the hardware and software synthesis more complicated than necessary.

For this reason, we define in this section the set of core statements that will be considered throughout the remaining chapters and sections. Statements that are not core statements are called macro statements, since the semantics of macro statements is defined as the semantics of a core statement that is used to define the macro statement.

However, it is not desired to work with a minimal set of core statements. Indeed, the set of core statements that is listed below contains still a lot of redundancy as will be explained later on. For example, it is not required to retain all of the four versions of the abortion or suspension statements. However, by considering the four variants in the following sections, their differences become more visible and clear.

Finally, we do not add the nondeterministic statements like the choose statement or the asynchronous or interleaving parallel statements to the core statements. Although these statements can not be replaced by equivalent deterministic statements, it is possible to do so when additional (controllable) input variables are added. In many languages including Esterel, such variables are called oracle variables.
Having these comments in mind, the core statements of Quartz are given in the following definition below:

**Definition 3.1. [Core Statements of Quartz]** The set of core statements of Quartz is the smallest set that satisfies the following rules, provided that $S$, $S_1$, and $S_2$ are also core statements of Quartz, $\ell$ is a location variable, $x$ is a local or output variable, $\sigma$ is a Boolean expression, and $\alpha$ a type:

- $x = \tau$ and next($x$) = $\tau$ (immediate/delayed assignment)
- assume($\sigma$); (assumption)
- assert($\sigma$); (assertion)
- nothing (empty statement)
- $\ell$: pause (consumption of time)
- if($\sigma$) $S_1$ else $S_2$ (conditional)
- $S_1; S_2$ (sequence)
- $S_1 \parallel S_2$ (synchronous concurrency)
- do $S$ while($\sigma$) (iteration)
- $\{ \alpha x; S \}$ (local declaration)
- during $S_1$ do $S_2$ (invariant action)
- abort $S$ when($\sigma$) (abortion)
- weak abort $S$ when($\sigma$) (weak abortion)
- immediate abort $S$ when($\sigma$) (immediate abortion)
- weak immediate abort $S$ when($\sigma$) (weak immediate abortion)
- suspend $S$ when($\sigma$) (suspension)
- weak suspend $S$ when($\sigma$) (weak suspension)
- immediate suspend $S$ when($\sigma$) (immediate suspension)
- weak immediate suspend $S$ when($\sigma$) (weak immediate suspension)

As already mentioned, the set of core statements is not minimal. Surprisingly, even the pause statement can be expressed by other statements as for example as follows:

$$\ell:\text{pause} \equiv \begin{cases} 
\text{abort} \\
\ell:\text{immediate suspend} \\
\text{nothing;} \\
\text{when(true)} \\
\text{when(true)} 
\end{cases}$$

Also the immediate preemption statements can be easily expressed by their delayed variants as follows:

- immediate abort $S$ when($\sigma$)
  $$\equiv \text{if}(\sigma) \text{ nothing;} \text{ else abort } S \text{ when}(\sigma);$$
- $\ell$:immediate suspend $S$ when($\sigma$)
  $$\equiv \text{while}(\sigma) \ell:\text{pause}; \text{ suspend } S \text{ when}(\sigma);$$

Hence, we see that the set of core statements is not minimal, and could be reduced if wanted. However, to make the differences of the various kinds of preemption statements clearer, we retain all of them in the set of core statements.
3.4 Macro Statements

In the previous section, we have defined the core statements and stated that the remaining statements can be defined in terms of the core statements. In this section, we list those definitions that are very simple. For some other statements, however, we need global transformations to adapt the context of the statement. In particular, the nondeterministic statements require the introduction of controllable inputs (oracles). For this reason, we describe the more complex macro statements in the following subsections. Also the transformation of the exception handling statements \texttt{try(e) } S_1 \texttt{ catch(e) } S_2 \texttt{ requires to consider the context of the statements, so that their translation requires a full pass over the syntax tree. Finally, we consider moreover also statements or concepts that have not been listed in Table 3.2, but that have been previously used or that are still used in the Esterel language.

3.4.1 Simple Macro Statements

In this section, we list some simple definitions of macro statements in terms of the core statements listed in Definition 3.1. Most of these statements are also used by the Esterel language:

- \texttt{emit x} \equiv x = true
- \texttt{emit next(x)} \equiv \texttt{next(x)} = true
- \texttt{if(\sigma) } S \equiv \texttt{if(\sigma) } S \texttt{ else nothing;}

\begin{verbatim}
switch
    case(\sigma_1) do S_1
    case(\sigma_2) do S_2
    ...
    case(\sigma_n) do S_n
else S
\end{verbatim}

\begin{verbatim}
if(\sigma_1) S_1
else if(\sigma_2) S_2
...
else if(\sigma_n) S_n
else S
\end{verbatim}

The 'correctness' of the above definitions should be clear\textsuperscript{1}. The difference between the immediate and delayed versions of the \texttt{await} statement should now also become clear.

Many more simple macro statements can be thought of and whenever they are frequently used, it makes sense to add such new statements. A simple way to introduce preliminary macro statements is to use text preprocessors like \texttt{cpp} or \texttt{m4}.

\textsuperscript{1} Actually, correctness is not the right word here, since the semantics of the macro statements is defined by these definitions in terms of the semantics of the core statements.
3.4.2 Additional Wait-Statements

- \( \ell:\text{halt} \equiv \text{do } \ell:\text{pause; while(true)} \)
- \( \ell:\text{await}(\sigma) \equiv \text{do } \ell:\text{pause; while(!\sigma) or alternatively, } \)
  \( \ell:\text{await}(\sigma) \equiv \text{abort } \ell:\text{halt; when(\sigma)} \)
- \( \ell:\text{immediate await}(\sigma) \equiv \text{while(!\sigma) } \ell:\text{pause; or alternatively, } \)
  \( \ell:\text{immediate await}(\sigma) \equiv \text{immediate abort } \ell:\text{halt; when(\sigma)} \)

3.4.3 Additional Loops

- \( \text{while(\sigma) } S \equiv \text{if(\sigma) do } S \text{ while(\sigma)} \)
- \( \text{loop } S \equiv \text{do } S \text{ while(true)} \)
- \( \ell:\text{loop } S \text{ each}(\sigma) \equiv \text{loop abort } S \ell:\text{halt when(\sigma)} \)
- \( (\ell_1, \ell_2): \text{every(\sigma) } S \equiv \ell_1: \text{await}(\sigma); \ell_2: \text{loop } S \text{ each}(\sigma) \)

3.4.4 Additional During-Statements

- \( \text{immediate during } S_1 \text{ do } S_2 \equiv S_2 \); during \( S_1 \) do \( S_2 \)
- \( \text{final during } S_1 \text{ do } S_2 \equiv \{ \text{during } S_1 \text{ do } S_2 \}; S_2 \)
- \( \text{immediate final during } S_1 \text{ do } S_2 \equiv S_2 \); \{ \text{during } S_1 \text{ do } S_2 \}; S_2 \)

3.4.5 Let-Abbreviations

We have already explained on page 56 that abbreviations \( \text{let}(x=\tau) \ S \) can be eliminated by replacing all occurrences of the variable \( x \) by \( \tau \) in \( S \). If common expressions are shared, this is the most efficient way to eliminate the statement. Otherwise, an alternative to eliminate the statement by a local variable declaration is as follows:

\[
\text{let}(x=\tau) \ S \equiv \\
\{ \alpha \ x; \\
\quad x=\tau; \\
\quad \text{during } S \\
\quad \text{do } x=\tau; \\
\quad x=\tau; \\
\} 
\]

This means that during the execution of \( S \) including starting and termination points of time, the variable \( x \) has always the same value as the expression \( \tau \). Therefore, \( x \) is an abbreviation of \( \tau \), which is the original meaning of the statement. Note that for the local variable \( x \), the event storage mode is sufficient, so that no memory is required for this variable.
3.4.6 Conjunctively Active Parallel Execution

The different forms of parallel execution of threads $S_1 \parallel S_2$, $S_1 \parallel\parallel S_2$, and $S_1 \mid S_2$ are active as long as one of the threads $S_1$ or $S_2$ are active. In other words, these statements terminate as soon as the last of the threads $S_1$ and $S_2$ terminates. One can also define related statements $S_1 \& S_2$, $S_1 \&\& S_2$, and $S_1 \& S_2$ that behave as $S_1 \parallel S_2$, $S_1 \parallel\parallel S_2$, and $S_1 \mid S_2$, respectively, but that terminate as soon as the first of the threads $S_1$ and $S_2$ terminates.

Such statements are useful when equivalent algorithms of different runtimes are executed in parallel. As soon as the first algorithm terminates, the result is available and the other thread can be aborted. For example, $S_1 \& S_2$ can be defined as follows (the other variants are defined analogously):

\[
S_1 \& S_2 := \{\text{event kill;}
\text{weak abort}
\{S_1; \text{emit(kill)}\};
\}
\text{||}
\{S_2; \text{emit(kill)}\};
\text{when(kill)}
\]

Although the above macro definition is simple, it has the drawback that a new local variable \texttt{kill} is used. However, as this variable is an event variable, it does not require registers in hardware or memory in software, so that it is essentially eliminated during compilation. For this reason, the use of the local variable is almost for free (the same holds if the statement is contained in a schizophrenic statement).

3.4.7 Generic Sequence and Parallel Statements

The definition of the generic sequence and parallels statements is simple. In the parameter specification $(\alpha \ x=\tau \ldots \pi \ 	ext{where} \ \sigma)$, the type $\alpha$ must be of one of the types \texttt{nat<n>} or \texttt{int<n>}. The expressions $\tau$ and $\pi$ must be static expressions, so that the compiler can reduce these expressions to values $\llbracket \tau \rrbracket$ and $\llbracket \pi \rrbracket$ of type $\alpha$. The specified parameters are then the numbers of the following set

\[
I := \{i \in \mathbb{Z} \mid \llbracket \tau \rrbracket \leq i \leq \llbracket \pi \rrbracket \land \llbracket \sigma \rrbracket^i_x\}
\]

The body statement $S$ of the generic statement is then instantiated with these parameters, i.e., the variable $x$ is replaced with each of the values of $I$. The resulting statements are then combined to a sequence or parallel execution. Note that these operators are associative. We may however demand that the combinations are done by a left-to-right associativity.

If the values $\llbracket \tau \rrbracket$ and $\llbracket \pi \rrbracket$ require a larger range $n$ of the types \texttt{nat<n>} or \texttt{int<n>}, then the same rules apply as for assignments as explained in Section 2.1.2.
3.4.8 Delay Statements

In order to model real-time systems, we often want to specify that more than one unit of time is consumed. In principle, one could use a sequence of pause statements for that purpose, but for large quantities of time to be consumed, this is not efficient. A better solution is therefore to use a counter as follows:

\[
\ell : \text{await}(n) := \{ \text{nat}<n+1> c; \\
    c = 0; \\
    \text{do} \\
    \quad \text{next}(c) = c + 1; \\
    \quad \ell : \text{pause} \\
    \quad \text{while}(c \neq n) \\
\}
\]

The above version of await does not await for a condition \( \sigma \) to hold, but for the next \( n \) instants of time. Hence, it will definitely terminate, and is equivalent (apart from not using further locations) to a sequence of \( n \) pause statements. Hence, we could also use the simpler alternative definition:

\[
\ell : \text{await}(n) := \text{sequence}(\text{nat } c = 1..n) \ell : \text{pause};
\]

The difference between the two alternatives is that the latter one may be unrolled by the compiler to \( n \) pause statements so that \( n \) new locations are generated, while the above version is reduced to size0f\( (n+1) \) bits only, but requires a new local variable with memory to store these bits. Nevertheless the number of state variables grows with \( O(n) \) in the second solution, while it only grows with \( O(\log(n)) \) using the first alternative.

3.4.9 Abstractions

For the verification of programs, it is often desirable to ‘hide’ some uninteresting states that are not relevant for the specification to be verified. ‘Hiding’ a state means to redirect all transitions leading to this state to all of its successor states. If transition systems are used where the transitions can be labelled with consumptions of time, then the number of transitions can be maintained during this hiding step. An important application is found in real-time model checking \([169, \ldots, 171]\), where intermediate states like those that are introduced by the await\((n)\) statement are not of interest.

An important prerequisite for this reduction procedure is to mark the irrelevant states that can be hidden afterwards. To distinguish between relevant and irrelevant states, we simply emit a special signal hide to mark the current state as an irrelevant one that can be deleted before verification. To make this kind of abstraction more convenient, we introduce the following two statements:
\ell:abstract(\sigma) \equiv \begin{align*}
do & \ell:pause; 
\text{if}(!\sigma) \text{ emit(hide);} 
\text{while}(!\sigma);
\end{align*}

\ell:abstract(n) \equiv 
\begin{align*}
\{ & \text{nat}<n+1> c; \\
& c = 0; \\
& \text{do} \\
& \quad \text{next}(c) = c+1; \\
& \quad \ell:pause; \\
& \quad \text{if}(!\sigma) \text{ emit(hide);} \\
& \quad \text{while}(c!=n) \\
& \}
\end{align*}

\ell:abstract(\sigma) simply waits for the condition \sigma, and until this condition becomes true, the signal hide is emitted to mark the states to be neglected. Note that hide is neither emitted at starting time nor at termination time of \ell:abstract(\sigma).

In the previous section, we have introduced another version of the await statement \ell:await(n) that waits for \(n\) instants of time. In analogy to that, it is natural to also define the following statement \ell:abstract(n):

3.4.10 Nondeterminism and Asynchronous Concurrency

So far, we have still omitted the nondeterministic statements, i.e., the nondeterministic choice and the asynchronous and interleaved parallel execution of threads. As these statements have a nondeterministic control flow, it is not possible to map these statements to equivalent deterministic statements without further means. In the following, we show how these statements are reduced to the deterministic core statements with the help of additional (controllable) input variables. Since the introduced input variables are controllable, we can ask by supervisory control whether there exists a schedule for the execution so that some property holds. Moreover, since the new variables are inputs, they are generated by the environment, so that their values appear to be nondeterministic.

Definition 3.2. [Reducing Nondeterministic Statements] Using new controllable boolean input variables \(c, c1, \text{ and } c2\), the following equations are used
to reduce nondeterministic choice, asynchronous and interleaved parallel statements to conditional choice and synchronous concurrency, respectively:

- choose $S_1$ else $S_2$ := if(c) $S_1$ else $S_2$

  $\{\text{event run1,run2};$
  $\quad\text{during suspend}$
  $\quad\quad S_1$
  $\quad\quad\text{when(run1&run2& c)};$
  $\quad\quad\text{do emit(run1)};$

- $S_1 | S_2$ :=

  $\{\text{event run1,run2};$
  $\quad\text{during suspend}$
  $\quad\quad S_1$
  $\quad\quad\text{when(run1&run2&!c)};$
  $\quad\quad\text{do emit(run2)};$

- $S_1 \parallel S_2$ :=

  $\{\text{event run1,run2};$
  $\quad\text{during suspend}$
  $\quad\quad S_1$
  $\quad\quad\text{when(run1&run2&(c_1&c_2))};$
  $\quad\quad\text{do emit(run1)};$

  $\quad\text{\parallel}$
  $\quad\text{during suspend}$
  $\quad\quad S_2$
  $\quad\quad\text{when(run1&run2&!(c_1|c_2))};$
  $\quad\quad\text{do emit(run2)};$

Consider the first equation: Depending on the value of the new input variable $c$, we choose one of the two statements for execution. As input variables receive their values from the environment which is nondeterministic, it follows that due to the above replacement, we deterministically react to values that are nondeterministically generated by the environment.

In a similar way, we reduced the asynchronous and interleaved parallel statements to a synchronous one. For the interleaved statement, we only need one new input variable $c$ and declare two further local event variables run1 and run2. These local event variables hold iff the corresponding thread is active, but not terminating: Since the local event variables run1 and run2 are emitted by the during statement, they do neither hold at starting time nor at termination time of the threads. Hence, no suspension takes place at starting time or at termination time of one of the threads. The same holds if one of the
two threads is no longer active, whether this may be due to the instantaneous execution of the entire thread or due to an already terminated thread.

Note that the suspension conditions can only hold when both threads are active and none of them is currently terminating. Moreover, the suspension conditions are mutually exclusive since \( c \) is either true or false. Hence, if both threads are active, but not terminating, exactly one of them is executed.

In a similar way, we implement the asynchronous parallel statement. To this end, we employ two new controllable input variables \( c_1 \) and \( c_2 \) and the local event variables \( \text{run1} \) and \( \text{run2} \) as described for the interleaved parallel statement. Again, \( \text{run1} \) holds if and only if the control flow is already somewhere inside \( S_1 \) and \( S_1 \) is currently not terminating, and the analogous statement holds for \( \text{run2} \) and \( S_2 \). Hence, the suspension conditions can only hold when both threads are active and none of them is currently terminating. In contrast to the interleaved execution, the suspension conditions are however no longer mutually exclusive: Instead, there are three cases as shown in the table below that allow us to execute either both threads or exactly one of them:

<table>
<thead>
<tr>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_1 &amp; c_2 )</th>
<th>( \neg(c_1 | c_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

It has to be noted that the above macro definitions can be used as they stand, but that the local event variables \( \text{run1} \) and \( \text{run2} \) can be replaced by the control flow predicates \( (S_i) \land \neg \text{term}(S_i) \) (as explained in the next chapter) during compilation. Hence, we can compile these statements without local event variables and only need new input control variables to introduce the nondeterminism. On the other hand, the use of the local event variables \( \text{run1} \) and \( \text{run2} \) is not too expensive, since these variables do not require memory, and since potential reincarnations are all statically false.

Moreover, since there are only two possible cases for the execution of choose \( S_1 \) else \( S_2 \) and \( S_1 \| S_2 \), a single boolean variable \( c \) is sufficient, while for the reduction of \( S_1 \| S_2 \), we need to distinguish three possible cases. Instead of using two boolean variables \( c_1 \) and \( c_2 \) for the reduction of \( S_1 \| S_2 \), we could alternatively use a single variable \( c \) of type \( \text{nat}<3> \), and some non-overlapping conditions like \( \text{run1} \& \text{run2} \& (c==0) \) and \( \text{run1} \& \text{run2} \& (c==2) \) (in case \( c==1 \), both threads are executed).

Is there a more efficient reduction of generic statements?

### 3.4.11 Exception Handling

Besides the \texttt{abort} statements, Esterel provides also the \texttt{trap} statement. This statement allows a process to immediately abort itself with a certain message so that a surrounding context statement can react accordingly. Similar
3 Statements, Interfaces and Modules

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2/P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>try(e) {</td>
<td>{event e;</td>
<td>{event e;</td>
<td>{event e;</td>
</tr>
<tr>
<td></td>
<td>emit x;</td>
<td>[weak] abort emit e;</td>
<td>immediate abort emit x;</td>
<td>weak immediate abort emit x;</td>
</tr>
<tr>
<td></td>
<td>throw e;</td>
<td>emit e;</td>
<td>emit e;</td>
<td>emit e;</td>
</tr>
<tr>
<td></td>
<td>emit y;</td>
<td>emit y;</td>
<td>emit y;</td>
<td>emit y;</td>
</tr>
<tr>
<td></td>
<td>} catch(e) nothing;</td>
<td>when(e)</td>
<td>when(e)</td>
<td>when(e)</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{x}</td>
<td>{x,y}</td>
<td>⊥</td>
<td>{x,y}</td>
</tr>
</tbody>
</table>

Table 3.4. Comparing exception handling with abortion

statements exist in many programming languages, and the naming conventions of these mechanisms are different: While in Esterel an trap is exited and handled elsewhere, Java programs throw exceptions and catch them in a surrounding context. In ML, exceptions are raised and handled by a surrounding statement. In Quartz, we follow the Java naming conventions, and talk about exceptions that are thrown and caught.

In this section, we will therefore discuss the meaning of the exception handling constructs throw e to throw the exception e, and the statement try(e) S catch(e) S_e to catch an exception e that is thrown by the body statement S. In particular, we discuss the mapping of these constructs to the core statements we have already defined.

To this end, we first have to locally declare a boolean typed variable of event storage mode that is used to replace the exception. Clearly, throw e must then somehow be mapped to emit e, and try(e) S catch(e) S_e must somehow be mapped to an abortion statement. However, these mappings are not so simple, for the following reasons:

- Exception handlers are neither weak nor strong abortion, which can be seen by the comparison given in Table 3.4. Instead, when an exception is thrown in a macro step, then it rather corresponds with a goto statement in that throw e immediately redirects the control flow to the catch statement of a surrounding try statement.
- Nested exception handlers behave slightly different to nested abortion statements, as we explain in detail below.

Consider first the issue that the throw statement is some variant of a goto statement by the programs in Table 3.4: The behavior of program P1 is that it will first emit x, then it will throw the exception e so that it will immediately terminate (since the catch clause is empty) and therefore, it will not emit y. Delayed abortion statements like P2 and P3 do not have an effect, since their body statements are instantaneous. Hence, both P2 and P3 emit both x and y. Programs P4 and P5 are however immediate abortions, so that the abortion caused by the emission of e will take place. However, as P4 is a strong abortion, the strong preemption has to invalidate the actions of its body, so
that we end up in a contradiction (and hence, $P_4$ has no consistent semantics).
In contrast, $P_5$ is consistent: the only possible behavior is to instantaneously terminate and to output both $x$ and $y$ (and also $e$ which is however not seen outside of its scope).

As can be seen, no abortion statement is equivalent to the exception handling statement: while weak or strong abortion either executes all or none of the data actions at abortion time, the exception handler will execute some of them as determined by the position of the throw statement. The throw statement therefore has the ability to redirect the control flow in a similar way like a goto statement.

The second point is the difference of nested exception handler and nested abortion statements, which is a rather subtle issue. To explain this difference, consider the following example:

```plaintext
module T1(event &a) {
  try(e1) {
    try(e2) {
      throw e1; || throw e2;
    }
    catch(e2) nothing;
    emit a;
  }
  catch(e1) nothing;
}
```

When the statement is started, then it concurrently throws both exceptions $e_1$ and $e_2$. The intended effect of concurrent exceptions is that the one with the highest priority is taken, which is the exception with the outermost scope. In the above example, this is exception $e_1$, since the try statement that declares $e_2$ in nested in the try statement that declares $e_1$.

Hence, $a$ is not emitted. Of course, we would have the same behavior when we would replace $\text{throw } e_1; || \text{throw } e_2;$ by $\text{throw } e_1; \text{throw } e_2;$, however we would have a different behavior when we would replace the same statement by $\text{throw } e_2; \text{throw } e_1;$. In the Esterel literature (e.g. in [42]), the priority schema of traps is governed by the so-called trap-level of an exit statement in a certain context. The trap level of exit $t$ is thereby 0 if it directly occurs in the body $S$ of its surrounding trap $t$ in $S$ end statement, but not in the body of any other trap statement contained in $S$. In general, the trap level of an exit statement is the number of nestings of other trap statements that embrace the exit statement in the body $S$ of its surrounding trap statement. If traps are raised, the one with the highest trap level is the one that has highest priority. In older papers like in [45], the same behavior has been specified with SOS transition rules that have been endowed with the set $T$ of raised traps of a transition.

There is a clever trick that can be used to implement the exception handling statements by core statements as follows: assume the statement $\text{try}(e) \ S \ \text{catch}(e) \ S_e$ is inside the body of some other try statements that refer to the exceptions $E = \{e_1, \ldots, e_n\}$. Inside the body $S$ there may be
further locally defined traps that are however not of interest for the construction of our abortion condition. Hence, each of the exceptions \( E \cup \{e\} \) can be thrown in \( S \), and the exceptions contained in \( E \) may also be thrown outside \( S \). Then, we replace \( \text{try}(e) \ S \ \text{catch}(e) \ S_e \) with the following statement:

\[
\text{try}(e) \ S \ \text{catch}(e) \ S_e \\
\{\text{event } e; \\\n\text{weak immediate abort} \\\nS; \text{emit } e; \text{pause}; \\\n\text{throw } e; \\\n\text{when}(e \& \! e_1 \& \ldots \& \! e_n); \\\n\text{if}(e \& \! e_1 \& \ldots \& \! e_n) S_e; \\\n\}
\]

The first idea is to replace \( \text{throw } e \) in \( S \) by \( \text{emit } e; \ \text{pause} \); and to use a weak immediate abortion when \( e \) is present. Instead of executing \( \text{throw } e \), we then emit \( e \) so that the weak abortion takes place. Moreover, since the control flow is preliminary stopped at the introduced \text{pause} statements, no further micro steps are executed in the threads of the body statement \( S \) after the \( \text{throw } e \) statements. Hence, the overall effect is the same as if the control flow would have been redirected from the places where \( \text{throw } e \) has been executed to the end of the abortion statement. Note that the introduced \text{pause} statements do not change the control flow since they are never reached, i.e., no new states are introduced by these locations.

The second idea is to not simply use the exception as the abortion condition, but rather the exception with the additional conjuncts that no surrounding exception has been thrown. If a surrounding exception would have been thrown, we also weakly abort the body statement \( S \), but at an outer nested abortion statement which leads to the correct implementation of the intended priorities.

For the previous example, we obtain the following transformed statement using the starting condition \( st \):

\[
\{\text{event } e_1; \\\n\text{weak immediate abort} \\\n\{\text{event } e_2; \\\n\text{weak immediate abort} \\\n\text{emit } e_1; \text{pause}; \\\n\text{emit } e_2; \text{pause}; \\\n\text{when}(e_2 \& \! e_1); \\\n\}\text{if}(e_2 \& \! e_1) \text{nothing}; \\\n\text{emit } a; \\\n\text{when}(e_1) \text{if}(e_1) \text{nothing}; \\\n\}
\]
Note that the inner abortion statement does not abort its body and preliminary allows the threads of its body statement to enter the pause statements. For this reason, emit a is not executed. Moreover, the outer abortion statement aborts its body statement, and therefore, we obtain the same behavior as the nested exception handling statements.

Note that the notion of concurrently thrown exceptions is a rather subtle one. For example, the following variant of the previous module T1 also throws exceptions e1 and e2 at the same time with a different behavior:

```plaintext
module T2(event &a) {
  try(e1) {
    try(e2) {
      throw e2;
      } catch(e2) nothing;
      emit a;
      throw e1;
    } catch(e1) nothing;
  }
}
```

Module T2 will emit a, since the inner try statement will terminate, and then emit a; is executed before the exception e1 is thrown. However, all of these micro steps belong to the same macro step.

Moreover, if a body statement consists of several threads and only one throws an exception, then the other threads are weakly aborted, i.e., they execute all micro steps as if no abortion would take place. Here is a further example to illustrate this behavior:

```plaintext
module T3(event &a) {
  try(e1) {
    throw e1;
    ||
    {
      try(e2) throw e2; catch(e2) nothing;
      emit a;
    }
    } catch(e1) nothing;
  }
}
```

At starting time, both exceptions are thrown, but as the throw e1; statement is not inside the body of the try(e2) statement, it is ignored for the inner abortion. Therefore, the try(e2) statement terminates, and a is emitted.

Finally, note that weak abortion statements do also respect the priorities of traps, and follow the compositional semantics of the language. For this reason, we also have to adapt the abortion conditions of the weak abortion statements that are nested in a try statement. Strong abortion statements inside a try statement may abort throw statements, so that no exception is
thrown when the abortion takes place. In contrast, weak abortion statements retain the thrown exceptions, so that the exception handler must be called and no further execution after termination of the weak abort statement should take place. To illustrate this, consider the following two statements, where abortion statements are nested inside trap statements:

\[
\begin{array}{c|c}
\text{try(e1)} & \text{try(e1)} \\
\text{immediate abort} & \text{weak immediate abort} \\
S_2 & S_2 \\
\text{when(e2);} & \text{when(e2);} \\
S_1 & S_1 \\
\} & \} \\
\text{catch(e1)} & \text{catch(e1)} \\
S_3 & S_3
\end{array}
\]

Consider first the left hand side: if the condition \(e_2\) holds, then the inner abortion statement will abort its body. As it is a strong abortion, all actions will not take place including potential \(\text{throw e1}\) statements. Hence, regardless whether \(e_1\) is thrown in \(S_2\) or not, at that time the strong abort statement behaves as nothing, and hence, statement \(S_1\) will be started.

The version on the right hand side behaves differently: As the abortion is weak, the inner abortion statement will allow its body \(S_2\) to throw the exception \(e_1\). If at the same time \(e_2\) holds, the abortion takes place, but \(e_1\) is, of course still thrown. Therefore, the exception handling takes place, and hence \(S_1\) can not be executed. Consider the following concrete examples:

\[
\begin{array}{ll}
\text{module T4(event \&a,\&b)} \\
\text{emit a;} \\
\text{try(e1)} \\
\text{immediate abort} \\
\text{throw e1;} \\
\text{when(a);} \\
\text{emit b;} \\
\} & \text{module T5(event \&a,\&b)} \\
\text{emit a;} \\
\text{try(e1)} \\
\text{weak immediate abort} \\
\text{throw e1;} \\
\text{when(a);} \\
\text{emit b;} \\
\} \\
\text{catch(e1) nothing;} \\
\text{catch(e1) nothing;}
\end{array}
\]

The correct translation to core statements must also modify the abortion conditions of weak abortion statements inside \text{try} statements in the same way as done for the translation of \text{try} statement to weak abort statements. For the above examples, we therefore obtain the following results:
3.4.12 Micro Step Variables

In synchronous languages all variables have uniquely determined values for each macro step. In Esterel, there are moreover variables that can change their values within micro steps, so that these variables may have more than one value per macro step. These variables must not be used in different threads, since it can not be defined which value of the variable should be used in the different threads. However, for one sequential thread, such variables can be defined with a consistent semantics.

We claim that these ‘micro step variables’ can be replaced with usual locally defined memorized variables. To this end, we have to follow the thread that owns the micro step variable and have to forward the current intermediate value during the micro steps to the expressions that use the corresponding variable. If we end up with a pause statement, we add an assignment of the form \( \text{next}(x) = \tau \), where the expression \( \tau \) is the last expression we had for the micro step variable.

Currently, Quartz does not offer the micro step variables, so that programmers have to apply the described transformation on their own if Esterel programs are ported to Quartz.

3.4.13 Inout Variables

Besides inputs and outputs, Esterel also provides inout variables. These variables can be modified by actions of the module as well as by actions of the environment. Quartz does not provide such variables, but offers controllable input variables that can be used for similar purposes (namely to model open systems).

\footnote{Reincarnated local variables are viewed here as different variables, even though they share the same name.}
Formal Semantics

In the previous chapter, we have defined the syntax of Quartz statements and modules, and we have already discussed their semantics in an informal way. The discussed examples made already clear that there is a crucial need for a precise definition of the semantics of the language since the reactive behavior of statements can become remarkably complex. For this reason, we define in this chapter the semantics of the Quartz language in a formal way that is the basis for all compilation techniques as well as for verification.

This formal semantics is an operational semantics that follows Plotkin’s structural approach to operational semantics [180, 209], which lends itself well for the description of the semantics of synchronous languages [42, 253]. Hence, the formal semantics is given by structural operational semantics (SOS) rules. To simplify matters, we split the formalization of the semantics into two steps:

- The first step is the definition of SOS transition rules based on a complete knowledge of the current environment (including the outputs that are currently generated by the statement). These SOS transition rules formalize an algorithm that computes the following:
  - The SOS transition rules can be used to check the consistency of the considered environment with the reaction of the considered statement. In particular, if an immediate assignment $x = \tau$ is executed, then the local/output variable $x$ must have the value $\lbrack \tau \rbrack^6_{\mathcal{E}}$. Hence, we can justify the values of local and output variables of the current environment by the SOS transition rules.

Note, however, that this is only one part that has to be checked for the consistency of the immediate actions with the current environment. In addition, we have to check the reaction to absence, which means that every variable $x$ must have its default value if no immediate assignment on it is executed in the current macro step and no delayed assignment on it was executed in the previous macro step. The default value of an event variable is determined by the type of the variable, and the
default value of a memorized variable is its previous value. If there was no previous macro step, then it is also determined by the type of the variable.

- Second, the SOS transition rules compute the delayed actions that partially contribute to the definition of the environment of the next macro step. To this end, the SOS transition rules simply collect pairs \((x, v)\) where \(x\) is the variable that should receive the value \(v\) in the next macro step. The collected pairs are then forwarded to the execution of the next macro step, where the pairs are used to determine the values of the corresponding variables \(x\). Note that the values \(v\) are obtained by an evaluation in the current macro step, and that the execution of delayed actions has no effect on the current macro step.

- Third, the SOS transition rules compute the residual statement that has to be executed in the next macro step. Intuitively, the residual statement is the remainder of the statement that is left by the execution of the given statement in the current macro step. In the literature, different kinds of SOS rules have been considered, and there are variants that differ in this case. Instead of computing a residual statement, an alternative is to deal only with active control flow locations which has several technical advantages. Both alternatives are equivalent, and therefore the second alternative is often called a ‘haltset encoding’ of the original SOS rules rather than a different presentation of the semantics.

- The second part of the formal semantics is also given in an operational style, and we also choose the formalism of SOS rules for the formal presentation. We call these SOS rules SOS reaction rules to distinguish them from the SOS transition rules. In contrast to the SOS transition rules, the SOS reaction rules are used to compute for given inputs for one macro step of a module the corresponding outputs in that macro step, so that a complete variable assignment of this macro step is obtained that can be used for the SOS transition rules. Therefore, the ‘second part’ is actually the first step of an interpreter.

It is important that the SOS reaction rules describe a constructive algorithm to incrementally compute the outputs for a so-far incomplete environment that may only have known values for the inputs. This algorithm performs a fixpoint iteration that is based on a symbolic execution of the program where each data type is endowed with an additional value \(\perp\) to express that the actual value of a local/output variable in the current macro step is yet unknown. The algorithm then computes on the one hand pessimistically a ‘must-set’ of actions \(D_{\text{must}}\) that must be executed regardless what values will finally replace the preliminary \(\perp\) values, and on the other hand, the algorithm optimistically computes a ‘can-set’ of actions \(D_{\text{can}}\) that can be executed regardless what values will finally replace the preliminary \(\perp\) values. By definition, we therefore always have \(D_{\text{must}} \subseteq D_{\text{can}}\).
In order to compute the sets $D_{\text{must}}$ and $D_{\text{can}}$, we have to compute pessimistic and optimistic estimations of the conditions that lead to an instantaneous execution of the program. For this reason, we have to generalize the semantics of expressions by the unknown values ⊥. We consider this generalization in Section 4.1.

This second part of the semantics, i.e., the fixpoint computation of the current outputs is in the spirit of [43] and is commonly referred to as causality analysis. As we already known that there are programs that implement logical contradictions or nondeterministic behavior (recall the discussion of Figures 3.3-3.5), it is not possible to compute unique outputs for all programs. However, even in the case where the program has a unique behavior for each input, it is not possible to compute this unique reaction by means of the SOS reaction rules.

One may argue that this is a deficiency of the SOS reaction rules that has to be improved. However, there are programs that should be rejected even though they have a unique behavior. The reason is that this unique behavior is obtained rather by accident than by a constructive computation of the program. We clearly have to forbid the accidental behavior since it is not computed by an algorithm, and is instead based on guesses which would make the determination of outputs much more complex (and questionable).

The programs whose unique reaction can be computed by the SOS reaction rules are called causally correct or constructive programs. The definition of constructive programs depends, of course, not only on the program, but also on the SOS reaction rules that we formulate. There is some freedom by the definition of these rules [229, 230] that does not compromise the constructive computation of the outputs, but may influence the complexity of the algorithms. We will discuss this issue in Section 4.6.

In the following chapters, we will moreover find deeply rooted equivalences between the constructive programs that are defined in terms of the SOS reaction rules and the results of their compilation: We will see that ternary simulation [57] of the hardware circuits that are generated from the programs by the algorithms discussed in Chapter 6 is equivalent to the constructiveness of the program. Moreover, the constructiveness of the program is equivalent to the existence of dynamic schedules for the software that is generated from the programs by the software synthesis algorithms of Chapter 6.

In Section 4.5, we define so-called control-flow predicates that are in some sense a symbolic description of the SOS transition rules. These control-flow predicates can be recursively defined in terms of the syntax tree of the program, which simplifies the formal treatment of the semantics, for example in theorem provers like HOL [223]. Using the control flow predicates, one can moreover define the control flow in terms of a formula that serves as a transition relation that can be directly used for symbolic model checking. Third, the control flow predicates are an intermediate step to proved the correctness of the hardware synthesis. Finally, we will see how the SOS reaction and transi-
tion rules can be used to translate modules to equivalent extended finite state machines.

4.1 Evaluating Expressions in Incomplete Environments

For the formal treatment of the semantics, we first have to describe how environments are modeled. Since local variables can be reincarnated several times within a macro step, they may have many values in a macro step, depending on the number of their reincarnations in the current macro step. For this reason, we use a function $E$ that maps all known variables $x$ to nonempty lists $E(x)$ of values: For every input or output variable $x$, $E(x)$ has exactly one value which is the uniquely determined value of $x$ in the current macro step. For a local variable $x$, $E(x)$ may consist of several values $[v_0, \ldots, v_n]$ that correspond with the different scopes\(^1\) of $x$ that are left and entered in this macro step: the execution of the micro steps starts in a scope where $x$ has value $v_0$. Then, the scope will be left and re-entered $n$ times in the same macro step, and $x$ has the values $v_1, \ldots, v_n$ in these scopes (in this order).

The length of the list $E(x)$ can be statically bounded: For input and output variables $x$, the length of the list $E(x)$ is 1, and for a locally declared variable $x$, the length of $E(x)$ is the number of loops that are nested around the local declaration of $x$. Note that reincarnations of a local variable can only occur due to loops that enforce that the local declaration is once more entered in the same macro step.

For every macro step, we therefore use a uniquely determined function $E$ that provides the values of the variables. The SOS rules, however, describe micro steps that refer to a particular scope of the local variables. To this end, we additionally maintain a function $h$ that maps the variables $x$ to a nonnegative integer $h(x)$ that is called the reincarnation index of the variable $x$.

The functions $E$ and $h$ determine the current environment, and both are updated during the execution of a micro step. In more detail, we define for the pair $(E, h)$ a variable assignment $\xi$ as $\xi(x) := v_{h(x)}$ where $E(x) = [v_0, \ldots, v_n]$ holds. Using this variable assignment, we formally define the evaluation of expressions w.r.t. to incarnated contexts as follows:

**Definition 4.1 (Evaluation of Expressions w.r.t. Incarnations).** Given an environment $E$ that maps the variables $V$ to lists of type-consistent values, i.e., for every variable $x \in V$, we have $E(x) = [v_0, \ldots, v_n]$ with $v_i \in \llbracket \alpha \rrbracket$ where $\alpha$ is the type of $x$. Moreover, let $h : V \to \mathbb{N}$ be a function with $h(x) < |E(x)|$.

Then, the semantics $\llbracket \tau \rrbracket_E^h$ of an expression $\tau$ with respect to $E$ and $h$ is defined as $\llbracket \tau \rrbracket_E^h := \llbracket \tau \rrbracket_E$, where the variable assignment $\xi$ is defined as $\xi(x) := v_{h(x)}$ with $E(x) = [v_0, \ldots, v_n]$ and $\llbracket \tau \rrbracket_\xi$ is defined in Definition 2.5.

\(^1\) The value $v_0$ is sometimes called the value of $x$ in the depth, while the values $v_1, \ldots, v_n$ are called the surface values of $x$. 
The above definition already handles the schizophrenia problem in that it allows us to handle the reincarnated values of local variables. The causality problem requires a further generalization of the semantics of expressions that will be explained in the remainder of this section.

As already explained, the computation of the current values of local and output variables is done by a fixpoint computation that starts with incomplete environments that are completed during the fixpoint iteration. To this end, the domain $\llbracket \alpha \rrbracket$ of type $\alpha$ is endowed by a special value $\perp$ that means that we do not yet know the actual value of $\llbracket \alpha \rrbracket$.

For this reason, we have to define for all operators of Table 2.1 that can be used to construct expressions what their result is when one of their arguments is the unknown value $\perp$. In general, we could argue that for an operator $f$, the result $f(\tau_1, \ldots, \tau_n)$ is $\perp$ whenever one of the arguments $\tau_i$ evaluates to $\perp$ (in the semantics of functional languages, these functions are often called strict).

However, there are functions $f$ whose value $f(\tau_1, \ldots, \tau_n)$ does not depend on one or the other argument $\tau_i$. For examples, consider the equations in Table 4.1.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \land false = false$</td>
<td></td>
</tr>
<tr>
<td>$false \land x = false$</td>
<td></td>
</tr>
<tr>
<td>$x \lor true = true$</td>
<td></td>
</tr>
<tr>
<td>$true \lor x = true$</td>
<td></td>
</tr>
<tr>
<td>$false \rightarrow x = true$</td>
<td></td>
</tr>
<tr>
<td>$x \rightarrow true = true$</td>
<td></td>
</tr>
<tr>
<td>$(x?y:y) = y$</td>
<td></td>
</tr>
<tr>
<td>$(true?x:y) = x$</td>
<td></td>
</tr>
<tr>
<td>$(false?x:y) = y$</td>
<td></td>
</tr>
<tr>
<td>$x \times 0 = 0$</td>
<td></td>
</tr>
<tr>
<td>$0 \times x = 0$</td>
<td></td>
</tr>
<tr>
<td>$(\tau &lt; 0) = true$ for any expression $\tau$ of type $\text{nat&lt;n&gt;$</td>
<td></td>
</tr>
<tr>
<td>$(\tau &lt; 0) = false$ for any expression $\tau$ of type $\text{nat&lt;n&gt;$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1. Laws for Lazy Evaluation of Expressions

Hence, in the spirit of lazy evaluation [124, 254] (also borrowed from functional languages), we could argue that there is no need to evaluate all arguments. Instead, it is sufficient to know the values of sufficiently many arguments to produce the result $f(\tau_1, \ldots, \tau_n)$.

This is an important key to enhance causality analysis. By the lazy evaluation rules, we can stop the propagation of the $\perp$ values, so that more output and local variables become known values even though they were unknown before. Of course, local and output values may also become known without the lazy evaluation rules of Table 4.1.
Definition 4.2 (Semantics of Expressions in Incomplete Environments). Environments \((\mathcal{E}, h)\) where there is a variable \(x\) such that the list \(\mathcal{E}(x)\) contains the unknown value \(\bot\) are called incomplete. The evaluation of an expression \(\tau\) in an incomplete environment \((\mathcal{E}, h)\) is defined as given in Definition 4.1 with the following exception: If one of the arguments of a function evaluation is \(\bot\), then the rules of Table 4.1 have to be applied to generate a value. If these rules cannot be applied, the result is \(\bot\).

Hence, using incomplete environments \((\mathcal{E}, h)\) can lead to the evaluation \(\llbracket \tau \rrbracket^E_h = \bot\) of an unknown value. This is often referred to as the ‘three-valued’ evaluation of expressions, since it is often considered in the domain of boolean values only, while we have described it already for arbitrary domains.

The three-valued evaluation of expressions is used in the fixpoint algorithm that aims to incrementally complete so-far incomplete environments. As we have to make sure that this fixpoint algorithm terminates, we define a partial order on incomplete environments as follows:

Definition 4.3 (Partial Order on Incomplete Environments). For incomplete environments \((\mathcal{E}, h)\) and \((\mathcal{E}', h')\) over the set of variable \(\mathcal{V}\), we define a partial order as follows:

\[(\mathcal{E}, h) \sqsubseteq (\mathcal{E}', h') :\iff \forall x \in \mathcal{V}. \mathcal{E}(x) \leq \mathcal{E}'(x),\]

where the partial order \(\leq\) on lists of values \([u_0, \ldots, u_m]\) and \([v_0, \ldots, v_n]\) is defined as follows:

\[ [u_0, \ldots, u_m] \leq [v_0, \ldots, v_n] \quad \iff \quad \begin{cases} m < n \\ m = n \text{ and } u_i = \bot \text{ implies } v_i = \bot \text{ for all } i \in \{0, \ldots, n\} \end{cases} \]

We will see that the function whose fixpoint is computed during the causality analysis is continuous with respect to the above partial order \(\sqsubseteq\). Thus, the well-known fixpoint theorem of Tarski and Knaster [251] guarantees the existence of the fixpoint, and we will be even more able to guarantee the termination of the fixpoint analysis since the underlying lattice has only finite chains.

4.2 SOS Transition Rules

Having explained the formal description of environments \((\mathcal{E}, h)\), we are ready to explain the first part of the formal semantics, i.e., the SOS transition rules. These SOS transition rules are of the form

\[(\mathcal{E}, h, S) \rightarrow (h', S', D, t)\]

with the following meaning:
4.2 SOS Transition Rules

- \((\mathcal{E}, h)\) is the complete environment of the current macro step.
- \(h'\) is the updated incarnation level function that is obtained by executing local variable declarations in \(S\).
- \(S'\) is the residual statement that has to be executed in the next micro or macro step (depending on the macro step flag \(t\)).
- \(D\) is a set of pairs \((x, v)\) where \(x\) is a variable and \(v\) is a value consistent with the type of \(x\). We have to assign the value \(v\) to \(x\) before starting the next macro step, since \(v\) is the value of \(x\) in the macro step that has been determined in the current macro step by delayed actions.
- Finally, the boolean value \(t\) denotes whether the execution described by the SOS transition rule is instantaneous. Hence, the SOS transition rule describes a micro step if \(t\) is true, and a macro step otherwise.

SOS rules for Esterel have been discussed in many papers including [42, 248, 253] and many equivalent variants exist. For example, some variants differ in terms of the modeling of trap levels etc. Moreover, the version of SOS rules we consider here determine a residual statement that is executed in the next (micro or macro) step. There are other variants that are based on the ‘halset encoding’ of the residual statement where only the set of active locations is stored instead of the residual statement. We consider this variant in Section 4.5.4.

The SOS transition rules that define the first part of the semantics of Quartz are given in Figures 4.1-4.5. It can be easily proved that for rules \(\langle \mathcal{E}, h, S \rangle \rightarrow \langle h', S', D, t \rangle\) that describe micro steps (i.e., with \(t = \text{true}\) so that \(S\) is instantaneous), the residual statement \(S'\) is always \text{nothing}. In the following, we give some remarks on the rules where we already make use of this invariant:

- The rule for an immediate assignment \(x = \tau\) requires that \(x\) and \(\tau\) evaluate to the same value in the considered environment \((\mathcal{E}, h)\). Otherwise, the execution is not possible since the environment is not consistent with the statement. Note that this rule does not allow the execution of programs with write conflicts or where a variable would be assigned a value different to the one that is provided by the current environment. For the next micro step, the residual statement \text{nothing} is left.
- The rule for delayed assignments simply collects the pair \((x, \llbracket \tau \rrbracket^h_D)\), so that the value \(\llbracket \tau \rrbracket^h_D\) can be used to partially determine the environment for the variable \(x\) in the next macro step. As in case of immediate assignments, the residual statement \text{nothing} is left to be executed in the next micro step.
- \(\text{assume}(\sigma)\) checks whether \(\sigma\) holds in the current environment. If this is the case, then it behaves like \text{nothing}, otherwise no execution is possible. Interestingly, \(\text{assert}(\sigma)\) has the same SOS rule, so that \(\text{assume}(\sigma)\) and \(\text{assert}(\sigma)\) share the same operational semantics. However, their meaning is different in terms of verification.
- \text{nothing} is always instantaneous and does neither modify the current nor the following environment. Hence, we have \(h' := h\), \(D := \{\}\), and \(t = \text{true}\).
\[ \{x\}^h_{\mathcal{E}} = \|\tau\|^h_{\mathcal{E}} \]
\[ (\mathcal{E}, h, x = \tau) \rightarrow (h, \text{nothing}, \{\}, \text{true}) \]
\[ (\mathcal{E}, h, \text{next}(x) = \tau) \rightarrow (h, \text{nothing}, \{(x, \|\tau\|^h_{\mathcal{E}})\}, \text{true}) \]
\[ \sigma^h_{\mathcal{E}} = \text{true} \]
\[ (\mathcal{E}, h, \text{assume}(\sigma)) \rightarrow (h, \text{nothing}, \{\}, \text{true}) \]
\[ \sigma^h_{\mathcal{E}} = \text{true} \]
\[ (\mathcal{E}, h, \text{assert}(\sigma)) \rightarrow (h, \text{nothing}, \{\}, \text{true}) \]
\[ (\mathcal{E}, h, \text{nothing}) \rightarrow (h, \text{nothing}, \{\}, \text{true}) \]
\[ (\mathcal{E}, h, \ell : \text{pause}) \rightarrow (h, \text{nothing}, \{\}, \text{false}) \]
\[ \sigma^h_{\mathcal{E}} = \text{true} \text{ and } (\mathcal{E}, h, S_1) \rightarrow (h_1, S_1', D_1, t_1) \]
\[ (\mathcal{E}, h, \text{if}(\sigma) \ S_1 \text{ else } S_2) \rightarrow (h_1, S_1', D_1, t_1) \]
\[ \sigma^h_{\mathcal{E}} = \text{false} \text{ and } (\mathcal{E}, h, S_2) \rightarrow (h_2, S_2', D_2, t_2) \]
\[ (\mathcal{E}, h, \text{if}(\sigma) \ S_1 \text{ else } S_2) \rightarrow (h_2, S_2', D_2, t_2) \]
\[ (\mathcal{E}, h, S_1) \rightarrow (h_1, S_1', D_1, \text{false}) \]
\[ (\mathcal{E}, h, \{S_1; S_2\}) \rightarrow (h_1, \{S_1'; S_2\}, D_1, \text{false}) \]
\[ (\mathcal{E}, h, S_1) \rightarrow (h_1, \text{nothing}, D_1, \text{true}) \]
\[ (\mathcal{E}, h, S_2) \rightarrow (h_2, S_2', D_2, t_2) \]
\[ (\mathcal{E}, h, \{S_1; S_2\}) \rightarrow (h_2, S_2', D_1 \cup D_2, t_2) \]
\[ (\mathcal{E}, h, S_1) \rightarrow (h_1, S_1', D_1, t_1) \]
\[ (\mathcal{E}, h, S_2) \rightarrow (h_2, S_2', D_2, t_2) \]
\[ (\mathcal{E}, h, \{S_1 \parallel S_2\}) \rightarrow (\text{Max}(h_1, h_2), \{S_1' \parallel S_2\}, D_1 \cup D_2, t_1 \land t_2) \]
\[ (\mathcal{E}, h, S) \rightarrow (h', S', D, \text{false}) \]
\[ (\mathcal{E}, h, \text{do } S \text{ while}(\sigma)) \rightarrow (h', \{S'; \text{while}(\sigma) \ S\}, D, \text{false}) \]
\[ \sigma^h_{\mathcal{E}} = \text{false} \]
\[ (\mathcal{E}, h, \{\text{while}(\sigma) \ S\}) \rightarrow (h, \text{nothing}, \{\}, \text{true}) \]
\[ \sigma^h_{\mathcal{E}} = \text{true} \text{ and } (\mathcal{E}, h, S) \rightarrow (h', S', D, \text{false}) \]
\[ (\mathcal{E}, h, \{\text{while}(\sigma) \ S\}) \rightarrow (h', \{S'; \text{while}(\sigma) \ S\}, D, \text{false}) \]
\[ (\mathcal{E}, h, h^{(h_{\mathcal{E}})}(x) + 1, S) \rightarrow (h', S', D, t) \]
\[ (\mathcal{E}, h, \{\alpha x; \ S\}) \rightarrow (h', S', \{(y, v) \in D \mid x \neq y, t\}) \]

Fig. 4.1. SOS Transition Rules (Part I)
The rule for $\ell:\text{pause}$ simply states that $\ell:\text{pause}$ does neither modify variables in the current nor in the next macro step. Instead, it terminates the current macro step which is indicated by the termination flag $t = \text{false}$. Moreover, nothing is left to be executed in the next macro step.

We have two rules for conditionals if($\sigma$) $S_1$ else $S_2$, depending on whether the condition $\sigma$ evaluates to true or false. Both rules are straightforward: depending on $\llbracket \sigma \rrbracket$, the condition either behaves exactly like $S_1$ or $S_2$.

For sequences $S_1; S_2$, we consider two cases. In each case, we first execute $S_1$ in the current environment $(E, h)$. The first rule now considers the case where the execution of $S_1$ is not instantaneous so that the control flow gets caught in the residual statement $S'$. In this case, $S_2$ is not started in the current macro step and the execution in the next macro step executes the sequence $S'; S_2$. The updates of the incarnation indices and the set of delayed actions are the same as for the partial execution of $S_1$.

The second rule considers the case where the execution of $S_1$ is instantaneous (recall that in this case, the residual statement is nothing by our invariant). For this reason it is sufficient to execute $S_2$ in the environment $(E, h_1)$ in the next micro steps. Clearly, the condition for instantaneous execution of $S_1; S_2$ is here the corresponding condition of $S_2$, the final incarnation indices are those obtained by the execution of $S_2$ and the set of executed delayed actions is the union of those executed by $S_1$ and $S_2$.

We have listed only one rule for $S_1 \parallel S_2$. This rule states that we execute both $S_1$ and $S_2$ in the environment $(E, h)$ yielding tuples $(h_i, S_i, D_i, t_i)$ for $i \in \{1, 2\}$. Clearly, we take the union of the delayed actions, and the condition for instantaneous execution of $S_1 \parallel S_2$ is the conjunction $t_1 \land t_2$.

For the remaining statement $S'_1 \parallel S'_2$, we have to remark that nothing $\parallel S'_2$ and $S'_1 \parallel$ nothing are both replaced with nothing in order to remove the parallel statement after termination of at least one thread.

The definition of the final incarnation level function also needs explanation: The execution of $S_i$ updates $h$ to $h_i$ in that the incarnation levels of the locally declared variables whose scopes are re-entered during execution of $S_i$ are increased according to the number of times these scopes have been re-entered. Now observe that $S_1$ and $S_2$ cannot re-enter the same local declaration since this would imply that this local variable would have been declared in both substatements. Hence, the updates of $S_1$ and $S_2$ refer to disjoint sets of local variables. The merge operation for incarnation level functions $\text{Max} (h_1, h_2)$ is therefore defined as follows for all variables $x$:

$$\left(\text{Max}(h_1, h_2)\right)(x) := \max(\{h_1(x), h_2(x)\})$$

The behavior of a do-while loop in the first macro step is defined as the execution of its body statement. However, when the remaining statement $S'$ will finally terminate, we have to check whether another iteration of
the loop has to be executed. This is conveniently done by adding a while-loop as a sequence to $S'$. However, this makes it necessary to also list SOS transition rules for while-loops, even though we do not view these as core statements.

- For the execution of while-loops, we first evaluate the loop condition $\sigma$. Depending on whether $\sigma$ holds or not, the loop behaves as nothing or as its loop body. In the latter case, another iteration may follow when the remainder $S'$ terminates, and therefore, we add a sequence with the while-loop.

Note that this rules yields a problem for formal reasoning about the SOS rules since the definition of the semantics of \texttt{while($\sigma$) S} by the larger statement \{\texttt{$S'$;while($\sigma$) S}\} contradicts a well-founded ordering along the syntax trees [224]. Instead, the SOS rules provide an operational semantics where a potentially infinite execution is defined and the progress is obtained on the micro steps that determine this infinite execution. Considering infinite traces would require more sophisticated induction techniques like co-induction. To simplify matters, the alternative approach by means of control flow predicates has been chosen to describe the semantics in [223] which has been proved to be equivalent to the SOS rules [224].

- The rule for executing a local declaration \{\texttt{$\alpha$ x; S}\} first increases the incarnation level of $x$ so that the updated incarnation level function $[h^{|h(x)|+1}]x$ is obtained from $h$. In this new environment, we execute the body statement $S$ of the local declaration which may furthermore update the incarnation level function to $h'$. Note that the updated incarnation level function $h'$ has to be used for further execution so that a further re-entering of \{\texttt{$\alpha$ x; S}\} considers a higher scope. Note further that we remove all delayed actions on the locally declared variable $x$ since they do no longer exist after leaving the scope.

- Figure 4.2 shows the SOS rules for the four versions of the abortion statement. Since the delayed variants are not active in the first macro step, we simply execute their body statement. If this immediately terminates, the abortion statement is simply ignored, otherwise, we embed the remaining statement $S'$ in a weak or strong immediate abortion context so that the abortion will be checked in all of the further macro steps.

Hence, it remains to explain the immediate abortion statements. If no abortion takes place, we consider the two cases as described for the delayed abortion, i.e., if the body statement instantaneously terminates, then the entire statement behaves as the body statement, and otherwise, we embed the remaining statement $S'$ in the corresponding abortion context. Finally, if the abortion of an immediate abortion statement takes place, we have to distinguish between the strong and the weak abortion. In case of strong abortion, the entire statement behaves as nothing, i.e., no actions of the body statement $S$ are executed. Instead, a weak abortion statement first executes its body statement $S$ thus yielding a tuple $(h', S', D, t)$. As the
4.2 SOS Transition Rules

\[ \langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]

\[ \langle E, h, S \rangle \rightarrow \langle h', S', D, \text{false} \rangle \]
\[ \langle E, h, \text{abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{immediate abort } S' \text{ when(}\sigma\text{)}, D, \text{false} \rangle \]

\[ \sigma^E_h = \text{true} \]
\[ \langle E, h, \text{immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h, \text{nothing}, \{\}, \text{true} \rangle \]

\[ \sigma^E_h = \text{false and } \langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]

\[ \sigma^E_h = \text{false and } \langle E, h, S \rangle \rightarrow \langle h', S', D, \text{false} \rangle \]
\[ \langle E, h, \text{immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{immediate abort } S' \text{ when(}\sigma\text{)}, D, \text{false} \rangle \]

\[ \langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{weak abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{no}\text{thing}, D, \text{true} \rangle \]

\[ \langle E, h, S \rangle \rightarrow \langle h', S', D, \text{false} \rangle \]
\[ \langle E, h, \text{weak abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{weak immediate abort } S' \text{ when(}\sigma\text{)}, D, \text{false} \rangle \]

\[ \sigma^E_h = \text{true and } \langle E, h, S \rangle \rightarrow \langle h', S', D, t \rangle \]
\[ \langle E, h, \text{weak immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]

\[ \sigma^E_h = \text{false and } \langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{weak immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]

\[ \sigma^E_h = \text{false and } \langle E, h, S \rangle \rightarrow \langle h', S', D, \text{false} \rangle \]
\[ \langle E, h, \text{weak immediate abort } S \text{ when(}\sigma\text{)} \rangle \rightarrow \langle h', \text{weak immediate abort } S' \text{ when(}\sigma\text{)}, D, \text{false} \rangle \]

Fig. 4.2. SOS Transition Rules (Part II)
\[ \langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{suspend } S \text{ when } (\sigma) \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, \text{suspend } S \text{ when } (\sigma) \rangle \rightarrow \langle h', \text{immediate suspend } S' \text{ when } (\sigma) \rangle, D, \text{false} \]
\[ \langle E, h, [\text{immediate suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h, [\text{immediate suspend } S \text{ when } (\sigma)] \rangle, \{\}, \text{false} \]
\[ \langle E, h, [\text{immediate suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \]
\[ \langle E, h, [\text{immediate suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h', \text{immediate suspend } S' \text{ when } (\sigma) \rangle, D, \text{false} \]
\[ \langle E, h, [\text{weak suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h', \text{weak immediate suspend } S' \text{ when } (\sigma) \rangle, D, \text{false} \]
\[ \langle E, h, [\text{weak immediate suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h, [\text{weak immediate suspend } S \text{ when } (\sigma)] \rangle, D, \text{false} \]
\[ \langle E, h, [\text{weak immediate suspend } S \text{ when } (\sigma)] \rangle \rightarrow \langle h', [\text{weak immediate suspend } S' \text{ when } (\sigma)] \rangle, D, \text{false} \]

Fig. 4.3. SOS Transition Rules (Part III)
abortion takes place, we ignore \( S' \) and \( t \) and replace them with nothing and true, respectively. In contrast to the strong abortion, we retain \( D \) since a weak abortion statement allows the execution of the actions of its body that would have been executed without the abortion.

- Figure 4.3 shows the SOS rules for the four versions of the suspension statement. The explanations of the delayed suspension statements are the same as in case of the abortion statements. For the immediate suspension statements, the cases where no suspension takes place are also explained as in case of the abortion statements.

Hence, it remains to explain the two rules for immediate suspension when the suspension condition \( \sigma \) holds. Both versions of the immediate suspension statement do then not instantaneously terminate. For this reason, they are also able to hold the control flow similar to pause statements (in fact, pause can be defined in terms of immediate suspension, as we already have mentioned). A strong suspension statement does not modify the current output variables, while a weak suspension statement allows the execution of the actions of its body statement as if no suspension would take place. In both cases, the current statement is also the remaining statement to be executed in the next macro step.

\[
\begin{align*}
\langle E, h, S_1 \rangle & \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \\
\langle E, h, \text{during } S_1 \text{ do } S_2 \rangle & \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \\
\langle E, h, S_1 \rangle & \rightarrow \langle h', S'_1, D, \text{false} \rangle \\
\langle E, h, \text{during } S_1 \text{ do } S_2 \rangle & \rightarrow \langle h', \text{during } S_1 \text{ do } S_2, D, \text{true} \rangle \\
\langle E, h, S_1 \rangle & \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \\
\langle E, h, \text{during } S_1 \text{ do } S_2 \rangle & \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle \\
\langle E, h, S_1 \rangle & \rightarrow \langle h', S'_1, D, \text{false} \rangle \\
\langle E, h, S_2 \rangle & \rightarrow \langle \text{nothing}, h'', S'_2, D_2, \text{true} \rangle \\
\langle E, h, \text{during } S_1 \text{ do } S_2 \rangle & \rightarrow \langle h', S'_1, D_1 \cup D_2, \text{false} \rangle
\end{align*}
\]

Fig. 4.4. SOS Transition Rules (Part IV)

- Finally, Figure 4.5 presents the SOS transition rules for the during statement. If \( S_1 \) is instantaneous, then during \( S_1 \) do \( S_2 \) simply behaves like \( S_1 \), i.e., \( S_2 \) is not executed. In case \( S_1 \) is not instantaneous, we make use of a pseudo-statement during \( S_1 \) do \( S_2 \) which may be viewed as an immediate form of during \( S_1 \) do \( S_2 \): While during \( S_1 \) do \( S_2 \) does not execute \( S_2 \) at starting time, during \( S_1 \) do \( S_2 \) does so provided that \( S_1 \) is not instantaneous at that point of time. Hence, if \( S_1 \) is not instantaneous, and leaves a residual \( S'_1 \) to be executed in the next macro step, then during \( S_1 \) do \( S_2 \) behaves like \( S_1 \) in the current macro step and leaves
the residual during $S_1$ do $S_2$ for the next macro step. It therefore remains to define the semantics of during $S_1$ do $S_2$, which is given by the last two rules of Figure 4.5: In case $S_1$ is instantaneous, during $S_1$ do $S_2$ behaves like $S_1$ (and therefore like during $S_1$ do $S_2$). Otherwise, $S_2$ is also executed and its actions are added to the actions executed by during $S_1$ do $S_2$. Note that the final rule demands that the execution of $S_2$ must be instantaneous.

\[
\begin{align*}
\text{[}x\text{]}_\mathcal{E} & = \text{true} \\
\langle \mathcal{E}, \mathcal{h}, \text{emit } x ; \rangle & \rightarrow \langle \mathcal{h}, \text{nothing}, \{ \}, \text{true} \rangle \\
\langle \mathcal{E}, \mathcal{h}, \text{emit next}(x) ; \rangle & \rightarrow \langle \mathcal{h}, \text{nothing}, \{(x, \text{true})\}, \text{true} \rangle \\
\langle \mathcal{E}, \mathcal{h}, \ell : \text{halt} \rangle & \rightarrow \langle \mathcal{h}, \ell : \text{halt}, \{ \}, \text{false} \rangle \\
\langle \mathcal{E}, \mathcal{h}, \ell : \text{await}(\sigma) \rangle & \rightarrow \langle \mathcal{h}, \ell : \text{immediate await}(\sigma), \{ \}, \text{false} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{[}\sigma\text{]}_\mathcal{E} & = \text{false} \\
\langle \mathcal{E}, \mathcal{h}, \ell : \text{immediate await}(\sigma) \rangle & \rightarrow \langle \mathcal{h}, \ell : \text{immediate await}(\sigma), \{ \}, \text{false} \rangle \\
\text{[}\sigma\text{]}_\mathcal{E} & = \text{true} \\
\langle \mathcal{E}, \mathcal{h}, \ell : \text{immediate await}(\sigma) \rangle & \rightarrow \langle \mathcal{h}, \text{nothing}, \{ \}, \text{true} \rangle \\
\langle \mathcal{E}, \mathcal{h}, \text{loop } S \rangle & \rightarrow \langle \mathcal{h}', \{ S' ; \text{loop } S \}, \mathcal{D}, \text{false} \rangle
\end{align*}
\]

Fig. 4.5. Derived SOS Transition Rules for Some Macro Statements

The SOS transition rules can already be used to prove what the behavior of statements within a complete environment. In particular, we can use the definitions of the macro statements to derive the corresponding SOS transition rules of the macro statements. For example, Figure 4.5 lists some transition rules of the simple macro definitions.

Moreover, we can use the SOS transition rules to prove the equivalence of statements. Let us consider a few simple examples:

- We defined \texttt{halt} as \texttt{do pause; while(true)}. Instead, we could also define it by \texttt{loop pause; or while(true) pause;}, since the latter lead to the same SOS transition rules.
- We already remarked that \texttt{pause} can be defined by an immediate suspend statement. This can now be formally proved in that the considered suspension statement leads to the same SOS transition rule. To this end, consider first the following application of the SOS transition rules:
It remains to prove that the obtained residual statement is behaviorally equivalent to `nothing`. This follows directly from the following instance of one of the SOS transition rules for strong immediate abortion:

\[
\langle E, h, \begin{cases} \text{immediate abort} \\
\text{nothing;}
\text{when(true)}
\end{cases} \rangle \rightarrow \langle h, \text{nothing}, \{\}, \text{true} \rangle
\]

Hence, also `nothing` can be easily defined in terms of an immediate abortion.

4.3 SOS Reaction Rules (Causality Analysis)

The first part of the semantics as formalized by the SOS transition rules of the previous section requires a completely known environment \((E, h)\). As the reactive system reads all inputs at the beginning of each macro step, the values of all input variables are known. Moreover, the values of those output and local variables are known that have been used as left hand sides of delayed assignments. However, all other output and local variables are unknown and are only determined by the current reaction itself.

In this section, we therefore complete the semantics in that we present an algorithm to iteratively complete an incomplete environment by computing the correct values of all local and output values. To this end, the algorithm performs a fixpoint iteration that starts with values \(\perp\) for the unknown output variables. For this reason, we make use of the three-valued evaluation of expressions as described in Definition 4.2.

In each iteration, the algorithm computes pessimistic and optimistic approximations \(D_{\text{must}}\) and \(D_{\text{can}}\) of the set of actions that are executed in the current macro step. Due to the determined actions \(D_{\text{must}}\) that must take place, or the information that no action can modify the variable (since no action on that variable belongs to \(D_{\text{can}}\)), some of these values become known. For this reason, the fixpoint iteration determines a finite sequence of environments \(E_0, E_1, \ldots, E_n\) that is a monotonically increasing chain with respect to the partial order described in Definition 4.3. Hence, in \(E_{i+1}\) at least one more variable became known compared to \(E_i\).

The fixpoint iteration terminates if no further progress is seen on the environments, thus the number of iteration steps is bounded by the number of variables that start with an unknown value \(\perp\). A program is causally correct if
for all output and local variables, known values are derived for all reachable states and all possible inputs.

The description of the fixpoint iteration is given in more detail in the next section. In this section, we concentrate on the computation of the pessimistic and optimistic approximations \( D_{\text{must}} \) and \( D_{\text{can}} \) that can be used to refine the so-far known environment \( E_i \). To this end, we describe this computation also with SOS rules that we call SOS reaction rules to distinguish them from the SOS transition rules of the previous section. These SOS reaction rules have the following form:

\[
\langle E, h, S \rangle \xrightarrow{\sigma} \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle,
\]

where \( D_{\text{must}} \) and \( D_{\text{can}} \) are the actions that \textit{must} be and \textit{can} be executed in the current environment and where the flags \( t_{\text{must}} \) and \( t_{\text{can}} \) denote whether the execution \textit{must} be or \textit{can} be instantaneous, respectively.

Due to the optimistic and pessimistic views, we can easily see that the conditions \( D_{\text{must}} \subseteq D_{\text{can}} \) and \( t_{\text{must}} \rightarrow t_{\text{can}} \) must invariantly hold. If the program is causally correct and the fixpoint iteration terminates, then the opposite directions do also hold, i.e., we then have \( D_{\text{must}} = D_{\text{can}} \) and \( t_{\text{must}} = t_{\text{can}} \).

The SOS reaction rules to determine the computation of the must/can information are given in Figures 4.6 and 4.7. The rules are more or less self-explanatory. For this reason, we only list a few remarks on some of these rules:

- For the atomic statements, the optimistic and pessimistic variants do not differ, i.e., we have \( D_{\text{must}} = D_{\text{can}} \) and \( t_{\text{must}} = t_{\text{can}} \) in the rules of these statements. In case of the immediate assignments, we only collect the statement in the sets \( D_{\text{must}} \) and \( D_{\text{can}} \) together with the current incarnation level function so that we are able to later on evaluate the right hand side expression \( \tau \). Delayed assignments can be ignored since they do not have an effect on the current reaction. Their effect on the next macro step is considered by the SOS transition rules. Also, the SOS reaction rules ignore \texttt{assume} and \texttt{assert} statements, since they also do not contribute to modifications of the current local/output variables. The validity of the conditions \( \sigma \) of \texttt{assume} and \texttt{assert} statements is checked by the transition rules (where the complete environment is known). There is nothing to remark for \texttt{nothing} and \texttt{pause}.

- Whenever the condition \( \sigma \) of conditional statement is evaluated to either true or false, we either execute the ‘then’ or ‘else’ branch of the statement which determines the result for the conditional statement. However, if the condition \( \sigma \) can not be evaluated to a boolean value, then \( D_{\text{can}} \) is the union of the can-sets obtained from the ‘then’ and the ‘else’ branches, while \( D_{\text{must}} \) is the intersection of the corresponding sets.

As we discuss in more detail in Section 4.6, this rule is already an improvement of Esterel’s view of causality as outlined in [229]. Our version of the conditional is sometimes also called McCarthy’s parallel if-then-else [64]
\(\langle \mathcal{E}, h, x = \tau \rangle \iff \langle \mathcal{E}, h, \{(x = \tau, h)\}, \{(x = \tau, h)\}, \text{true}, \text{true} \rangle\)

4.3 SOS Reaction Rules (Causality Analysis)

\(\langle \mathcal{E}, h, \text{next}(x) = \tau \rangle \iff \langle \mathcal{E}, h, \{\}, \{\}, \text{true}, \text{true} \rangle\)

\(\langle \mathcal{E}, h, \text{assume}(\sigma) \rangle \iff \langle \mathcal{E}, h, \{\}, \{\}, \text{true}, \text{true} \rangle\)

\(\langle \mathcal{E}, h, \text{assert}(\sigma) \rangle \iff \langle \mathcal{E}, h, \{\}, \{\}, \text{true}, \text{true} \rangle\)

\(\langle \mathcal{E}, h, \text{nothing} \rangle \iff \langle \mathcal{E}, h, \{\}, \{\}, \text{true}, \text{true} \rangle\)

\(\langle \mathcal{E}, h, \ell : \text{pause} \rangle \iff \langle \mathcal{E}, h, \{\}, \{\}, \text{false}, \text{false} \rangle\)

\([\sigma]_h^0 = \text{true} \quad \langle \mathcal{E}, h, S_1 \rangle \iff \langle h_1, D_{\text{must}}, D_{\text{can}}^1, t_{\text{must}}, t_{\text{can}}^1 \rangle\)

\(\langle \mathcal{E}, h, \text{if}(\sigma) S_1 \ \text{else} \ S_2 \rangle \iff \langle h_1, D_{\text{must}}, D_{\text{can}}^1, \text{true}, \text{true} \rangle\)

\([\sigma]_h^1 = \text{false} \quad \langle \mathcal{E}, h, S_2 \rangle \iff \langle h_2, D_{\text{must}}, D_{\text{can}}^2, t_{\text{must}}, t_{\text{can}}^2 \rangle\)

\([\sigma]_h^2 = \bot \quad \langle \mathcal{E}, h, \text{if}(\sigma) S_1 \ \text{else} \ S_2 \rangle \iff \langle h_2, D_{\text{must}}, D_{\text{can}}^2, t_{\text{must}}, t_{\text{can}}^2 \rangle\)

\(\langle \mathcal{E}, h, S_1 \rangle \iff \langle h_1, D_{\text{must}}, D_{\text{can}}^1, \text{false}, \text{false} \rangle\)

\(\langle \mathcal{E}, h, S_1 \rangle \iff \langle h_1, D_{\text{must}}, D_{\text{can}}^1, \text{false}, \text{true} \rangle\)

\(\langle \mathcal{E}, h, S_1 \rangle \iff \langle h_2, D_{\text{must}}, D_{\text{can}}^2, t_{\text{must}}, t_{\text{can}}^2 \rangle\)

\(\langle \mathcal{E}, h, S_2 \rangle \iff \langle h_2, D_{\text{must}} \cup D_{\text{can}}^2, t_{\text{must}} \cup t_{\text{can}}^2, t_{\text{can}}^2 \rangle\)

\(\langle \mathcal{E}, h, S \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\([\| \sigma \|_h^0 = \text{false} \quad \langle \mathcal{E}, h, \text{while}(\sigma) \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\([\sigma]_h^1 = \text{true} \quad \langle \mathcal{E}, h, S \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\([\| \sigma \|_h^1 = \text{false} \quad \langle \mathcal{E}, h, \text{while}(\sigma) \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\(\langle \mathcal{E}, h, S \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\(\langle \mathcal{E}, h, \text{next}(\alpha + S) \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)

\(\langle \mathcal{E}, h, \text{SOS Reaction Rule Start} \rangle \iff \langle h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle\)
\[ \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{[weak] immediate suspend } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \llbracket \sigma \rrbracket_h^b = \text{false} \quad \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{[weak] immediate abort } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \llbracket \sigma \rrbracket_h^b = \text{true} \quad \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{weak immediate abort } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, \text{true, true}) \]
\[ \llbracket \sigma \rrbracket_h^b = \text{true} \quad \langle E, h, \text{immediate abort } S \text{ when}(\sigma) \rangle \triangleright (h, \text{[{}]}, \text{true, true}) \]
\[ \llbracket \sigma \rrbracket_h^b = \perp \quad \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{weak immediate abort } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, \text{true, true}) \]
\[ \llbracket \sigma \rrbracket_h^b = \perp \quad \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{immediate abort } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, \text{false, false}) \]
\[ \llbracket \sigma \rrbracket_h^b = \text{true} \quad \langle E, h, \text{immediate suspend } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \llbracket \sigma \rrbracket_h^b = \text{false} \quad \langle E, h, S \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}) \]
\[ \langle E, h, \text{weak immediate suspend } S \text{ when}(\sigma) \rangle \triangleright (h', D_{\text{must}}, D_{\text{can}}, \text{false, true}) \]

Fig. 4.7. SOS Reaction Rules (Part II)
in contrast to the sequential if-then-else of Esterel. The explanation of taking the maximum of the incarnation level functions is exactly the same as given on page 91 in the case of parallel statements.

- Sequences \{S_1; S_2\} introduce the uncertainty to estimate the instantaneity of the first part \(S_1\): Therefore, we consider three cases, namely the first case where \(S_1\) is definitely known to be not instantaneous, the second case where nothing is known about the instantaneity of \(S_1\), and the third case where the execution of \(S_1\) must be instantaneous. These three cases are handled by the three SOS rules of \{S_1; S_2\}. In the first rule, \(S_1\) cannot be instantaneous since \(t_{\text{can}} = \text{false}\). Therefore, we can only execute \(S_1\) and not \(S_2\), and therefore all approximations are obtained from \(S_1\) alone.

In the second rule, \(S_1\) may not be instantaneous, but it can be so, hence, we are not sure about the instantaneity of \(S_1\). Therefore, the actions that must be executed are those that must be executed by \(S_1\), while the actions that can be executed are the union of the actions that can be executed by \(S_1\) or \(S_2\). As \{S_1; S_2\} can be instantaneously executed iff both \(S_1\) and \(S_2\) can be instantaneously executed, \(t_{\text{can}} = t_{\text{can}}^2\) in this case.

Finally, in the third rule, we know that the execution of \(S_1\) must be instantaneous since \(t_{\text{must}} = \text{true}\). Therefore, we must also execute \(S_2\), and the statements that must and can be executed are the unions of the corresponding sets of \(S_1\) and \(S_2\). The approximation of the instantaneity is then determined by \(S_2\).

- The rule for the parallel statement is obvious and needs no further explanation.
- The approximations of the do-while loop are simply obtained from the body statement, since that is definitely executed.
- The cases, where the loop condition evaluates to a boolean value are clear: We either execute nothing or the loop body. However, if the loop condition is unknown, then the while loop introduces a further source of uncertainty. In this case, the must-approximation assumes that no further iteration will take place, while the can-approximation assumes another iteration. Therefore, the final \(D_{\text{must}}\) is empty, while the final \(D_{\text{can}}\) is the result of the can-approximation of the body statement. The instantaneity is unclear in this case, which means that \(t_{\text{must}} = \text{false}\) and \(t_{\text{can}} = \text{true}\).

- The rule for the local declaration is simple. It only increases the incarnation level of the locally declared variable so that the further analysis refers to the right incarnation.
- The rule for the during \(S_1\) do \(S_2\) statement is also simple, since this statement executes in the current reaction only its first substatement \(S_1\). If \(S_1\) is not instantaneous, then the SOS transition rules of the during statement will generate a sequence with \(S_2\) that we need not consider in the SOS reaction rules.
- The rules for abortion statements are given in the upper half of Figure 4.7. The first four rules define the cases, where we are sure whether an abor-
tion takes place (either due to delayed abortion statement or due to the fact that the abortion condition evaluates to a boolean value).

Finally, the last two rules introduce again uncertainty, since we are not sure whether the immediate abortion takes place or not: Since the abortion can take place, we conclude that the execution can be instantaneous, thus we have \( t_{can} = \text{true} \). On the other hand, we can not deduce that the execution must be instantaneous due to the abortion, since there is still the chance that \( \sigma \) could change to false. Hence, the only case where the execution will unavoidably be instantaneous is when the execution of the abortion statement’s body must be instantaneous.

- The rules for suspension statements are given in the lower half of Figure 4.7. Analogously to the abortion statements, the first four rules define the cases, where we are sure whether a suspension takes place (either due to delayed suspension statement or due to the fact that the suspension condition evaluates to a boolean value).

The last two rules are again the difficult cases. These rules introduce once more uncertainty, since we have an immediate suspension, but are not sure about the value of the suspension condition. Since the suspension can take place (since \( \sigma \) could change to true), we define \( t_{must} = \text{false} \), since even in the case where the suspension statement’s body must be instantaneous, it may still be the case that the suspension leads to a non-instantaneous execution. The execution can be instantaneous if the body statement can be instantaneous.

In a rule \( \langle \mathcal{E}, h, S \rangle \rightarrow \langle h', D_{must}, D_{can}, t_{must}, t_{can} \rangle \), the flags \( t_{must} \) and \( t_{can} \) may also be viewed to encode one of the three values 1, 0, and \( \perp \), when we assume the encoding of these values as given by the leftmost table below:

<table>
<thead>
<tr>
<th>( t_{must} )</th>
<th>( t_{can} )</th>
<th>&amp;</th>
<th>( | )</th>
<th>( \lor )</th>
<th>( x \land \neg x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>true</td>
<td>0</td>
<td>1</td>
<td>⊤</td>
</tr>
<tr>
<td>⊥</td>
<td>true</td>
<td>true</td>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>⊤</td>
<td>true</td>
<td>true</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
</tr>
</tbody>
</table>

The other three tables define the three-valued versions of conjunction, disjunction, and negation. Having this view, we already work with a three-valued logic to analyze the instantaneousity condition of the statement’s execution. We will consider this view in more detail in Section 4.5, where we compute a boolean formula for the instantaneousity of a statement.

### 4.4 Complete Operational Semantics

Based on the SOS rules given in the two previous sections, we are now able to implement an interpreter for causally correct Quartz programs. Figure 4.8 es-
function MacroStep(E_{prev}, D_{prev}, S)

  // preparation phase
  E_{new} = \text{ReadInputs}();
  E_{new} = E_{new} \cup \{(x, [\bot]) \mid x \in V_{out} \cup V_{loc}\};
  ℏ_{init} = \{(x, 0) \mid x \in V_{in} \cup V_{out} \cup V_{loc}\};

  // perform delayed actions of previous macro step
  for (x, v) \in D_{prev} do
    E_{new} = \begin{cases} E_{new} ; & E_{new} = E_{new} \cup \{\text{default}(x)\} \end{cases};
  end;

  // start fixpoint iteration to incrementally complete E_{new}
  do
    E_{old} = E_{new};
    // compute must/can components
    \langle ℏ_{new}, D_{must}, D_{can}, t_{must}, t_{can} \rangle \leftarrow \langle E_{old}, ℏ_{init}, S \rangle;
    // react to absence of actions on event variables
    for x \in V_{event} \cup V_{mem} do
      H = \{h(x) \mid (x = \tau, h) \in D_{can}\};
      if 0 \not\in H \land \begin{cases} E_{new} = \bot \end{cases} then
        E_{new} = E_{new} \begin{cases} \text{default}(x) \end{cases};
      end;
    end;
    // react to absence of actions on memorized variables
    for x \in V_{event} \cup V_{mem} do
      H = \{h(x) \mid (x = \tau, h) \in D_{can}\};
      if 0 \not\in H \land \begin{cases} E_{new} = \bot \end{cases} then
        E_{new} = E_{new} \begin{cases} \text{default}(x) \end{cases};
      end;
    end;
    // react to absence of actions on incarnated local variables
    for x \in V_{event} \cup V_{mem} do
      for i = 1 \ldots ℏ_{new}(x) do
        H = \{h(x) \mid (x = \tau, h) \in D_{can}\};
        if i \not\in H \land \begin{cases} E_{new} = \bot \end{cases} then
          E_{new} = E_{new} \begin{cases} \text{default}(x) \end{cases};
        end
      end
    end;
    // execute immediate actions of D_{must}
    for (x = \tau, h) \in D_{must} do
      E_{new} = E_{new} \begin{cases} \text{default}(x) \end{cases};
    end;
  while (E_{old} \neq E_{new});

  // transfer value of highest reincarnated value to the depth
  for x \in V_{loc} \cup V_{mem} do
    E_{new} = E_{new} \begin{cases} \text{default}(x) \end{cases};
  end;
  // compute next statement
  \langle ℏ', S', D, t \rangle \leftarrow \langle E_{new}, ℏ_{init}, S \rangle;
  return (E_{new}, S', D, t);
end
Fig. 4.8. A Simple Interpreter for Quartz Statements
sentially gives the implementation of our interpreter that we explain in detail below.

The function MacroStep given in Figure 4.8 performs the computation of a macro step for a given statement \( S \). In addition to the statement \( S \), the function receives as arguments the environment \( E_{\text{prv}} \) of the previous macro step and the set of delayed actions \( D_{\text{prv}} \) that have been computed in the previous macro step (which is initially the empty set). The outputs of the function are the current environment \( E_{\text{new}} \), the residual statement \( S' \), the set of delayed actions \( D \) that are executed in the current macro step, and the termination flag \( t \) that tells us whether the program terminated or not.

The notation \( [E]_v \) means that the environment function \( E \) is changed for variable \( x \) so that the \( h \)-th value in the list \( E(x) \) is replaced with \( v \). Moreover, we simply write 0 for the incarnation level function \( h \) that maps every variable to 0. Thus, \( [x]_0^0 \) is the first value of \( E(x) \), i.e., \( x \)'s value in the depth.

In the preparation phase, we read in the new inputs and reset the values of all output and local variables to the unknown value \( \perp \). The incarnation level of all variables is set to zero in the initial incarnation level function \( h_{\text{init}} \) that will be used in each iteration (since we start in the depth and climb up some surfaces of scopes). Hence, every variable is mapped to a list of values that has the length 1. After this, we 'perform' the delayed assignments that have been executed in the previous macro step. This means we change the value of each variable \( y \) at level 0 to the value \( v \) for each pair \( (y, v) \) that occurs in \( D_{\text{prv}} \).

With the obtained environment \( E_{\text{new}} \), we now start the fixpoint iteration whose task is to compute the reaction of the program \( S \). To this end, we use the SOS reaction rules given in Figure 4.6 and Figure 4.7 to compute the tuple \( \langle h_{\text{new}}, D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle \) from the tuple \( \langle E_{\text{new}}, h_{\text{init}}, S \rangle \). The flags \( t_{\text{must}} \) and \( t_{\text{can}} \) are no longer of interest here; they have only been used to be able to perform the recursive computation of \( D_{\text{must}} \) and \( D_{\text{can}} \).

Using the can-approximation \( D_{\text{can}} \) we now first compute the reaction to absence. Intuitively, this means that we give every local/output event variable its default value if that variable has not yet a known value and if the variable can not be modified by an immediate action in the current macro step. Similarly, we assign to every memorized variable its previous value if that variable has not yet a known value and if the variable can not be modified by an immediate action in the current macro step. Third, we compute the reaction to absence for the reincarnations of local variables. These are treated as event type variables, since they only live for the current macro step, and therefore they are initialized with the default values unless an immediate action determines another value.

After the reaction to absence, some variables may already have changed their value from \( \perp \) to a known value in the updated environment \( E_{\text{new}} \). Using this updated environment \( E_{\text{new}} \), we now evaluate the expressions \( \tau \) in all immediate actions \( y = \tau \) that we collected in the previous computation of the set \( D_{\text{must}} \). This enables us to finally execute the immediate actions which may change the environment \( E_{\text{new}} \) once more. Note here that we need the
incarnation level function that refers to the scopes where the assignment \( x = \tau \) has been executed to be able to evaluate \( \tau \) correctly. Alternatively, we could have evaluated \( \tau \) during the computation of the SOS reaction rules, but then we could not benefit from the above reaction to absence, so that a further iteration would be required for some programs that is not necessary for the algorithm in Figure 4.8.

Finally, we check whether a progress in the fixpoint iteration has been obtained, i.e., if \( \mathcal{E}_{\text{new}} \) differs from \( \mathcal{E}_{\text{old}} \). If this is the case, we repeat the iteration in order to further approximate the fixpoint. In addition, we may check whether the environment \( \mathcal{E}_{\text{new}} \) is already complete, i.e., if all variables are mapped to values different to \( \bot \). If this should be the case, we know that a further iteration can not update \( \mathcal{E}_{\text{new}} \) furthermore, so that we can also stop the computation.

After the termination of the fixpoint iteration, we have determined the complete environment \( \mathcal{E}_{\text{new}} \) of the current reaction, i.e., in particular all values of the local and output variables in this macro step. Using \( \mathcal{E}_{\text{new}} \) and the initial incarnation level function \( h_{\text{init}} \), we now compute the tuple \( \langle h', S', D, t \rangle \) from the tuple \( \langle \mathcal{E}_{\text{new}}, h_{\text{init}}, S \rangle \) according to the rules given in Figures 4.1-4.3. Depending on \( t \), we may then invoke the computation of the next macro step or terminate the interpretation due to termination of the Quartz program.

The function whose fixpoint is computed is only implicitly given. It is a function from environments to environments and is used to compute the new value of \( \mathcal{E}_{\text{new}} \) from the previous value of \( \mathcal{E}_{\text{new}} \). It is easily seen that this function is monotonic, since during the fixpoint iteration only changes in the environments are done for variables that currently have the value \( \bot \), which increases the environment.

The termination of the fixpoint iteration that is done in function MacroStep is obvious, since our lattice of environments has only finite increasing chains. The reason for this is that every statement has only finitely many variables, and therefore every environment \( \langle \mathcal{E}, h \rangle \) is also defined for only these finitely many variables. In every fixpoint iteration, one of the local or output variables must change its value from \( \bot \) to a known value. Hence, this can only happen finitely often: The number of iterations is bounded by the number of local/output variables that have initially a value \( \bot \).

It may be the case that the fixpoint iteration ends with an environment that still contains unknown values \( \bot \) for some local/output variables. In this case, the program is not constructive and an error must be reported by the interpreter. Note that only constructive programs are considered to be semantically sound, and nonconstructive programs, even if they should have a unique behavior are viewed as not correct.
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

In the previous sections, we have described the operational semantics of the Quartz language in terms of SOS transition and SOS reaction rules. This has been given in full detail and can be used to compute the meaning of every constructive program. The formalization of the semantics in an operational style directly leads to an interpreter function as given in Figure 4.8.

However, compilers have to analyze the program for all possible inputs at compile time, so that the operational approach as given in the previous sections is only of limited use. Due to the presence of infinite data types, it is not possible to enumerate all cases, and even if only finite data types were used, this enumeration would be too expensive.

For this reason, a better approach is required to analyze all inputs at compile time. This is done by a symbolic treatment of the SOS transition and reaction rules that is described in this section. ‘Symbolic’ means here that we do work with environments that are updated as done in the interpreter. Instead, we construct ‘symbolic representations’ for the values, i.e., expressions of the corresponding type.

There is not much to do for program expressions like expressions $\sigma$ or $\tau$ that occur in the program except that we have to respect the current scopes of the local variables that is encoded in the incarnation level function $h$. To this end, we simply write $h(\sigma)$ to denote an expressions that is endowed with this information. Technically, this can be implemented by endowing the variable’s names with their incarnation level index, i.e., $x^{(h(x))}$ or by using a new operator to make this more explicit like $\text{reincar}(x, h(x))$. We abstract from the actual representation and simply write $h(\sigma)$ for the modified expression $\sigma$.

Moreover, we sometimes have to refer to next value of a variable $y$ that we denote as $\text{next}(y)$. Clearly, we must not refer to next values of inputs, but we may determine the next value of a local or output variable due to a delayed action. Moreover, we can determine the next value of location variables $\ell$ that uniquely refer to control flow locations of the program. These location variables are the state variables of the control flow automaton that we describe in this section, which is an extended finite state machine whose symbolic description is derived in this section.

Using this notation, it is then straightforward to define control flow predicates and guarded commands that are in some sense symbolic representations of the SOS transition rules. We will discuss the relationship between these symbolic descriptions and the SOS rules in more detail in Section 4.5.4.

In the following, we define the control flow predicates and the computation of the guarded commands in detail. A further advantage (in addition to the symbolic description) in comparison to the SOS transition rules is thereby that the definition of the control flow predicates as well as the definition of the guarded commands is done by simple primitive recursive definitions. Hence,
the termination of these computations is obvious, which drastically simplifies
the embedding of the semantics in theorem provers [223, 224].

Before presenting the precise definitions of the symbolic descriptions, we
first discuss the intuitive meaning in the following list:

\( \text{lbls}(S) \) is the set of locations that occur in statement \( S \). This is the set of
labels \( \ell \) of all statements that can held the control flow at some place like
\( \ell : \text{pause} \) or \( \ell : \text{await}(\sigma) \).

\( \text{in}(S) \) is the disjunction of the variables \( \text{lbls}(S) \). Therefore, \( \text{in}(S) \) holds at
some point of time if the control flow is currently at some location inside
\( S \), i.e., if \( S \) is active.

\( \text{inst}_h(S) \) holds iff the execution of \( S \) is instantaneous. Note that this property
depends on inputs, so that we usually compute an expression \( \text{inst}_h(S) \)
depending on the current values of input, local, and output variables.
Note that \( \text{inst}_h(S) \) does not depend on the current locations \( \text{lbls}(S) \), since
it is checked whether the control flow from outside \( S \) can go through \( S \)
without being caught in \( S \). Hence, it is assumed that \( S \) is currently not active.

\( \text{enter}_h(S) \) describes where the control flow will be caught when \( S \) would
now be started. Again, this depends on the current values of input, local,
and output variables. In addition to the input, local, and output variables,
the expression \( \text{enter}_h(S) \) moreover depends on the (next values of the)
location variables so that we also have occurrences \( \text{next}(\ell) \) for location
variables \( \ell \) in \( \text{enter}_h(S) \). However, location variables \( \ell \) do not occur with-
out the next operator, since it is assumed that \( S \) is currently not active,
and it is formalized where the control flow is caught next when \( S \) is now
started.

\( \text{term}(S) \) describes all conditions where the control flow is currently some-
where inside \( S \) and wants to leave \( S \). Note, however, that the control flow
might still be in \( S \) at the next point of time, since \( S \) may be (re)entered
at the same time, e.g., by a surrounding loop statement. The expression
\( \text{term}(S) \) therefore depends on input, local, output, and location variables
without next-operators.

\( \text{move}(S) \) describes all internal moves, i.e., all possible transitions from some-
where inside \( S \) to another location inside \( S \) without temporarily leaving
\( S \).

Guarded Actions: While the above predicates only consider the control flow,
the data flow is computed in two steps. The first step, we compute for
all actions of the program the corresponding preconditions that enable
the execution of the action. Guarded actions are pairs of the form \((\gamma, C)\),
where \( C \) is an action, i.e., an assignment, \( \text{assume} \) or \( \text{assert} \) statement. The
meaning of \((\gamma, C)\) is that \( C \) is immediately executed whenever its guard \( \gamma \)
holds.

In a second step, we then compute a transition relation of the guarded
actions. Clearly, this transition relation implies that for every guarded ac-
tion $(\gamma, C)$, the validity of the guard $\gamma$ implies the effect of the action $C$. In addition to this, we moreover have to implement the reaction-to-absence, in that we defined the values of the variables if no action is currently given them a value. A subtle problem is furthermore imposed by reincarnated variables: We finally have to transfer the value of the most recent incarnation to the actual local variable.

The control flow may only rest at the control flow locations of a statement $S$ that we have marked with location variables $\ell$. Hence, to define the control flow of $S$, it is sufficient to describe all situations where the control flow can move from the set of currently active locations to the set of locations that are active at the next point of time.

It is possible to compute the residual statement for a given set of active locations of a statement $S$. Therefore, we can recast the SOS transition rules in terms of sets of active locations instead of residual statements. This has already been done in [40, 42] and will be considered in Section 4.5.4 to establish the relationship between the above control flow predicates and the SOS transition rules. Intuitively, $\text{inst}_h(S)$ then describes the transitions from outside $S$ so that no location of $S$ will be active in the next step, while $\text{enter}_h(S)$ describes the transitions from outside $S$ so that some locations of $S$ will be active in the next step. Moreover, $\text{term}(S)$ describes the transition rules from somewhere inside $S$ to the empty haltset, while $\text{move}(S)$ describes the transition rules from somewhere inside $S$ to another place inside $S$.

### 4.5.1 Computing the Control Flow

In the following, we simply list the primitive recursive definitions of the mentioned control flow predicates. We start with the definition of $\text{inst}_h(S)$ which is given below:

**Definition 4.4 (Instantaneous Execution).** Given a Quartz statement $S$, the following recursively defined formula $\text{inst}_h(S)$ holds iff $S$ instantaneously terminates:

- $\text{inst}_h(x = \tau) \equiv \text{inst}_h(\text{next}(x) = \tau) \iff \text{true}$
- $\text{inst}_h(\text{assume}(\sigma)) \equiv \text{inst}_h(\text{assert}(\sigma)) \iff \text{true}$
- $\text{inst}_h(\text{nothing}) \equiv \text{true}$
- $\text{inst}_h(\ell : \text{pause}) \iff \text{false}$
- $\text{inst}_h(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) \iff \left( h(\sigma) \land \text{inst}_h(S_1) \lor \neg h(\sigma) \land \text{inst}_h(S_2) \lor \text{inst}_h(S_1) \land \text{inst}_h(S_2) \right)$
- $\text{inst}_h(S_1 ; S_2) \iff \text{inst}_h(S_1) \land \text{inst}_h(S_2)$
- $\text{inst}_h(S_1 \parallel S_2) \equiv \text{inst}_h(S_1) \land \text{inst}_h(S_2)$
- $\text{inst}_h(\text{do} \ S \ \text{while}(\sigma)) \equiv \text{inst}_h(S)$
- $\text{inst}_h(\{\alpha \ x ; S\}) \equiv \text{inst}_h(S)$
- $\text{inst}_h(\text{during} \ S_1 \ \text{do} \ S_2) \equiv \text{inst}_h(S_1)$
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

- \( \text{inst}_h ([\text{weak}] \text{ abort } S \ when(\sigma)) \Leftrightarrow \text{inst}_h (S) \)
- \( \text{inst}_h ([\text{weak}] \text{ immediate abort } S \ when(\sigma)) \Leftrightarrow h(\sigma) \lor \text{inst}_h (S) \)
- \( \text{inst}_h ([\text{weak}] \text{ suspend } S \ when(\sigma)) \Leftrightarrow \text{inst}_h (S) \)
- \( \text{inst}_h (\ell :[\text{weak}] \text{ immediate suspend } S \ when(\sigma)) \Leftrightarrow \neg h(\sigma) \land \text{inst}_h (S) \)

Note that a statement can be instantaneous for a certain environment and may consume time for another one. For example, for the following statement \( S \)

\[
\text{if}(x) \begin{cases} \\
\ell : \text{pause}; \\
\text{emit } y_1; \\
\text{emit } y_2;
\end{cases}
\]

we compute \( \text{inst}_h (S) \) as \( x \land \text{false} \lor \neg x \land \text{true} \lor \text{false} \land \text{true} \), and hence, equivalent to \( \neg x \). Thus, the statement can be instantaneous iff the current value of \( x \) is false.

As can be seen, only the conditions \( \sigma \) of if-statements, and the immediate preemption statements determine whether a statement is instantaneously executed. For all the other statements, the instantaneousity does not depend on the current environment.

Using \( \text{inst}_h (S) \), we next define \( \text{enter}_h (S) \) for a Quartz statement \( S \), which describes the set of all transitions that enter \( S \) from outside.

**Definition 4.5 (Entering Statements).** Given a Quartz statement \( S \), the following recursively defined formula \( \text{enter}_h (S) \) describes which control flow locations are activated by entering a statement \( S \):

- \( \text{enter}_h (x = \tau) \Leftrightarrow \text{enter}_h (\text{next}(x) = \tau) \Leftrightarrow \text{false} \)
- \( \text{enter}_h (\text{assume}(\sigma)) \Leftrightarrow \text{enter}_h (\text{assert}(\sigma)) \Leftrightarrow \text{false} \)
- \( \text{enter}_h (\text{nothing}) \Leftrightarrow \text{false} \)
- \( \text{enter}_h (\ell : \text{pause}) \Leftrightarrow \text{next}(\ell) \)
- \( \text{enter}_h (\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) \Leftrightarrow \left( \begin{array}{c} \text{enter}_h (S_1) \land \neg \text{next}(in(S_2)) \land h(\sigma) \lor \text{enter}_h (S_2) \land \neg \text{next}(in(S_1)) \land \neg h(\sigma) \end{array} \right) \)
- \( \text{enter}_h (S_1 ; S_2) \Leftrightarrow \left( \begin{array}{c} \text{enter}_h (S_1) \land \neg \text{next}(in(S_2)) \lor \text{enter}_h (S_2) \land \neg \text{next}(in(S_1)) \land \text{inst}_h (S_1) \end{array} \right) \)
- \( \text{enter}_h (S_1 \parallel S_2) \Leftrightarrow \left( \begin{array}{c} \text{enter}_h (S_1) \land \text{inst}_h (S_1) \land \neg \text{next}(in(S_2)) \lor \text{enter}_h (S_2) \land \text{inst}_h (S_2) \land \neg \text{next}(in(S_1)) \lor \text{enter}_h (S_1) \land \text{enter}_h (S_2) \end{array} \right) \)
- \( \text{enter}_h (\text{do } S \ \text{while}(\sigma)) \Leftrightarrow \text{enter}_h (S) \)
- \( \text{enter}_h (\{ x ; \ S \}) \Leftrightarrow \text{enter}_h (S) \)
- \( \text{enter}_h (\text{during } S_1 \ \text{do } S_2) \Leftrightarrow \text{enter}_h (S_1) \land \neg \text{in}(S_2) \)
- \( \text{enter}_h ([\text{weak}] \ \text{abort } S \ when(\sigma)) \Leftrightarrow \text{enter}_h (S) \)
- \( \text{enter}_h ([\text{weak}] \ \text{immediate abort } S \ when(\sigma)) \Leftrightarrow \neg h(\sigma) \land \text{enter}_h (S) \)
- \( \text{enter}_h ([\text{weak}] \ \text{suspend } S \ when(\sigma)) \Leftrightarrow \text{enter}_h (S) \)
• $enter_h(ℓ; \text{[weak] immediate suspend } S \ when(σ))$
  \[ \iff \left( \neg h(σ) \land enter_h(S) \land \neg next(ℓ) \lor \right) \]
  \[ \left( h(σ) \land \neg \text{in } (S) \land next(ℓ) \right) \]

For example, for the previously considered statement, we obtain the enter formula $next(ℓ) \land \neg \text{false} \land x \lor \neg next(ℓ) \land \text{false} \land \neg x$, i.e., the formula $next(ℓ) \land x$. Hence, if $x$ is true, then the control flow location $ℓ$ will be active at the next point of time, otherwise it location of the statement is active, since its execution is then instantaneous.

Again, only the conditions $σ$ of if-statements, and the immediate preemption statements determine the direction of the control flow at starting time. For all the other statements, the initial control flow does not dependent on the current environment.

Note further that we have to explicitly encode that when entering a substatement of an if-statement or a sequence that no location of the other substatement is active at the next point of time. We will therefore start to establish the invariant that of if-statements and sequences, at most one substatement is active. We do the same for the during statement, but since $\text{inst}_h(S_2)$ must be statically true, this can also be neglected. However, the exclusiveness has to be added for the immediate suspension statements.

We proceed with the definition of term $(S)$ that describes the set of transitions where the control flow leaves a statement $S$. For this reason, we assume that the control flow already rests somewhere inside $S$ and now wants to leave $S$.

**Definition 4.6 (Termination of Statements).** Given a Quartz statement $S$, the following recursively defined formula term $(S)$ describes all situations, where the control flow leaves $S$:

• $\text{term } (x \Rightarrow τ) :\iff \text{term } (\text{next } (x) \Rightarrow τ) :\iff \text{false}$
• $\text{term } (\text{assume } (σ)) :\iff \text{term } (\text{assert } (σ)) :\iff \text{false}$
• $\text{term } (\text{nothing}) :\iff \text{false}$
• $\text{term } (ℓ; \text{pause}) :\iff ℓ$
• $\text{term } (\text{if } (σ) S_1 \text{ else } S_2) :\iff \left( \text{term } (S_1) \land \neg \text{in } (S_2) \lor \right) \left( \text{term } (S_2) \land \neg \text{in } (S_1) \right)$
• $\text{term } (S_1; S_2) :\iff \left( \text{term } (S_1) \land \neg \text{in } (S_2) \land \text{inst}_0 (S_2) \lor \right) \left( \text{term } (S_2) \land \neg \text{in } (S_1) \lor \right) \left( \text{term } (S_1) \land \text{term } (S_2) \right)$
• $\text{term } (S_1 \text{ || } S_2) :\iff \left( \text{term } (S_1) \land \neg \text{in } (S_2) \lor \right) \left( \text{term } (S_2) \land \neg \text{in } (S_1) \lor \right) \left( \text{term } (S_1) \land \text{term } (S_2) \right)$
• $\text{term } (\text{do } S \text{ while } (σ)) :\iff \neg σ \land \text{term } (S)$
• $\text{term } (\{ α; x; S \}) :\iff \text{term } (S)$
• $\text{term } (\text{during } S_1 \text{ do } S_2) :\iff \text{term } (S_1)$
• $\text{term } (\text{[weak] abort } S \ when(σ)) :\iff \text{in } (S) \land σ \lor \text{term } (S)$
• $\text{term } (\text{[weak] immediate abort } S \ when(σ)) :\iff \text{in } (S) \land σ \lor \text{term } (S)$
• $\text{term } (\text{[weak] suspend } S \ when(σ)) :\iff \neg σ \land \text{term } (S)$
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

- \( \text{term}(\ell:\text{weak} \text{ immediate suspend } S \text{ when}(\sigma)) \)
  \[\iff\quad \neg \sigma \land \ell \land \text{in}(S) \land \text{inst}_{h_0}(S) \lor \neg \sigma \land \neg \ell \land \text{term}(S)\]

As can be seen, \( \text{term}(S) \) is false whenever the control flow is currently not inside \( S \). Hence, \( \text{term}(S) \) is false for the instantaneous atomic statements. For the same reason, we can easily prove that \( \text{term}(S) \rightarrow \text{in}(S) \) holds. However, as \( \text{term}(S) \) does only depend on the current values of the input, local, output, and location variables, it does not impose any restrictions on the next values of the location variables. Hence, it may be the case that some of the locations of \( S \) may be active at the next macro step, even though \( \text{term}(S) \) holds. This may happen due to a surrounding loop whose loop body terminates and is re-entered at the same macro step.

Note that the definition of \( \text{term}(S) \) invokes some calls to \( \text{inst}_{h}(S) \) that we endowed with the initial incarnation level function 0 that maps every variable to zero. This is done since the micro steps encoded by \( \text{term}(S) \) start from somewhere inside \( S \), i.e., in the depth of \( S \).

In case of the termination of sequences and if-statements we have added the redundant terms that when a substatement terminates then the other substatement is currently not active. Since we establish the invariant that at most one of the substatements of sequences and if-statements can be active at a point of time, this can be safely removed.

Finally, we define the control flow predicate \( \text{move}(S) \) that encodes all internal moves, i.e., transitions from somewhere inside \( S \) to possibly different locations inside \( S \). To this end, \( \text{move}(S) \) combines some of the micro steps encoded in \( \text{inst}_{h}(S) \), \( \text{enter}_{h}(S) \), and \( \text{term}(S) \). However, for the definition of the internal moves of the suspension statements, we need to say also that the current control flow is frozen, i.e., that the same control flow locations are active at the next point of time. This is formalized by the predicate \( \text{stutter}(S) \) that is defined as \( \text{stutter}(S) \equiv \bigwedge_{\ell \in \text{lbls}(S)} \ell = \text{next}(\ell) \).

**Definition 4.7 (Internal Moves).** Given a Quartz statement \( S \), the following recursively defined formula \( \text{move}(S) \) describes all conditions where the control flow moves inside \( S \):

- \( \text{move}(\text{x=}\tau) \iff \text{move}((\text{next}(\text{x=})=\tau) \iff \text{false} \)
- \( \text{move}(\text{assume}(\sigma)) \iff \text{move}(\text{assert}(\sigma)) \iff \text{false} \)
- \( \text{move}(\text{nothing}) \iff \text{false} \)
- \( \text{move}(\ell:\text{pause}) \iff \text{false} \)
- \( \text{move}(\text{if}(\sigma) \quad S_1 \quad \text{else} \quad S_2) \iff \begin{cases} \text{move}(S_1) \land \neg \text{in}(S_2) \land \neg \text{next}(\text{in}(S_2)) \lor & \text{move}(S_2) \land \neg \text{in}(S_1) \land \neg \text{next}(\text{in}(S_1)) \\ \text{term}(S_1) \land \neg \text{next}(\text{in}(S_1)) \land \neg \text{in}(S_2) \land \text{enter}_{h_0}(S_2) \end{cases} \)
- \( \text{move}(S_1;S_2) \iff \begin{cases} \text{move}(S_1) \land \neg \text{in}(S_2) \land \neg \text{next}(\text{in}(S_2)) \land \neg \text{next}(\text{in}(S_2)) \lor & \text{move}(S_2) \land \neg \text{in}(S_1) \land \neg \text{next}(\text{in}(S_1)) \lor \\ \text{term}(S_1) \land \neg \text{next}(\text{in}(S_1)) \land \neg \text{in}(S_2) \land \text{enter}_{h_0}(S_2) \end{cases} \)
form in that we combine these predicates.

Given a Quartz state-
notation, we define the micro steps that need to be considered for a definition of the entire control flow of a statement $S$. We will now define the control flow automaton in a symbolic form in that we combine these predicates.

To this end, we have to solve a further technical problem: We assume that at the initial point of time, the control flow is not yet in the statement, but intends to enter it. However, if the statement will finally terminate, then the control flow will also be outside the statement without the intention to enter it again. For this reason, we have to distinguish between the starting state and the termination state. To this end, we introduce a further location $st$ that is used to start the execution of the process when there is no control flow active in it. This location is called the initial location or the boot location.

In the definition of the control flow automaton below, we formalize that the initial location $st$ holds exactly at the initial point of time. The definition of the control flow automaton is then as follows:

**Definition 4.8 (Control Flow with Initial Location).** Given a Quartz statement $S$, and a the initial location $st$ that does not occur in $S$, we define the set of initial states and the transition relation of the control flow automaton of $S$ by the following formulas $I_{cf}(st, S)$ and $R_{cf}(st, S)$:
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

The control flow automaton is therefore a finite state machine whose states are encoded by the state variables $\{st\} \cup \text{lbls}(S)$. Transitions are labeled by conditions that are all encoded in the transition relation $R_{cf}(st, S)$. The automaton has only a single initial state, namely the one encoded by $I_{cf}(st, S)$. Starting from there, there are two possibilities: Either $S$ can be instantaneous, which is described in the first disjunct of $R_{cf}(st, S)$, or the control flow can enter $S$ which is described in the second disjunct. Once inside $S$, we may follow internal transitions (third disjunct), or the control flow might leave $S$ (fourth disjunct). Once outside the statement, it will never be re-entered, since the initial location is false after the initial point of time.

To prove the equivalence with the hardware circuits as described in Chapter 6, it is necessary to obtain a more flexible definition of the control flow in that the statement may be restarted after it has already been executed. To be more precise, the statement may be restarted when it either terminates or when there is not yet a control flow active in it. To this end, we do not use a initial location, but rather view $st$ to be an external signal that controls the initial start as well as a later restart (that is required for the verification of loops):

**Definition 4.9 (Control Flow with Start Signal).** Given a Quartz statement $S$, and a start signal $st$ that does not occur in $S$, we define the set of initial states and the transition relation of the control flow automaton of $S$ by the following formulas $I_{cf}(st, S)$ and $R_{cf}(st, S)$:

$$I_{cf}(st, S) :\iff st \land \neg \text{in}(S)$$

$$R_{cf}(st, S) :\iff \begin{cases} st \land \neg \text{in}(S) \land \text{inst}_{ho}(S) \land \neg \text{next}(\text{in}(S)) \lor \\ st \land \neg \text{in}(S) \land \text{enter}_{ho}(S) \lor \\ \neg st \land \text{move}(S) \lor \\ \neg st \land (\neg \text{in}(S) \lor \text{term}(S)) \land \neg \text{next}(\text{in}(S)) \end{cases} \land \text{next}(st)$$

Let us call a statement startable if $\neg \text{in}(S) \lor \text{term}(S)$ holds. Note that the start signal $st$ is only respected when the statement is startable, that is, an already active statement $S$ is not restarted unless it terminates. If it currently terminates, it may be restarted at the same instant of time.

Using the HOL theorem prover we have proved a couple of simple properties of the control flow predicates. These properties are important for any kind of formal reasoning about programs, in particular, they could be important for automatic proof procedures. The most important properties are summarized in the following lemma.

**Lemma 4.10 (Properties of Control Flow Predicates).** For any Quartz statement $S$, the following facts hold for the control flow predicates:
• enter_h(S) → next(in(S))
• enter_h(S) → ¬inst_h(S)
• term(S) → in(S)
• move(S) → in(S) ∧ next(in(S))
• move(S) → ¬term(S)
• stutter(S) → (in(S) ↔ next(in(S)))
• ¬in(S) → (stutter(S) = ¬next(in(S)))

The transition relation for the control flow has been given in a disjunctive form above which is convenient to split the different possible transitions into meaningful cases. Using the above properties, it is possible to prove the equivalence of this transition relation to a conjunctive form, which convenient in the dual cases. For example, if R_{cf}(st,S) appears as an assumption in a goal to be proved, then one can obtain the same meaningful case distinctions by the conjunctive version of the control flow below.

**Lemma 4.11 (Conjunctive Form of Control Flow Transition Relation).** For any Quartz statement S and the start signal st, the transition relation R_{cf}(st,S) of the control flow of S is equivalent to the following formula:

\[
\begin{align*}
&\left( ((¬\text{in}(S) \lor \text{term}(S)) \land \text{st} \land \text{inst}_h(S) \rightarrow ¬\text{next}(\text{in}(S))) \land \\
&\quad ((¬\text{in}(S) \lor \text{term}(S)) \land \text{st} \land ¬\text{inst}_h(S) \rightarrow \text{enter}_h(S)) \land \\
&\quad ((¬\text{in}(S) \lor \text{term}(S)) \land ¬\text{st} \rightarrow ¬\text{next}(\text{in}(S))) \land \\
&\quad (\text{in}(S) \land ¬\text{term}(S) \rightarrow \text{move}(S))
\right)
\]

For the proof of the above lemma, we have to recall that for sequences, if-statements and immediate suspensions, at most one substatement may be active. Hence, for \(S \equiv \text{if } \sigma \text{ then } S_1 \text{ else } S_2 \text{ end}\) or \(S \equiv S_1; S_2\), we can derive from the assumption in(S), and the fact that the transition relation R_{cf}(st,S) always holds, that in(S_1) ≠ in(S_2) holds.

Using these facts, it is even possible to prove a recursive computation schema for the transition relation as shown in the following theorem. Note, however, that we still need the definitions of the control flow predicates in(S), inst_h(S), and term(S) for this purpose. However, we can avoid the computation of enter_h(S) and move(S):

**Theorem 4.12 (Recursive Definition of Control Flow).** For any Quartz statement S, the transition relation R_{cf}(h, st, S) can be recursively computed according by the following laws (starting with h = 0), provided that the assumption st → ¬in(S) ∨ term(S) holds, i.e., that statements are only started when they are not active or are currently terminating:

• \(R_{cf}(h, st, x = \tau) \Leftrightarrow R_{cf}(h, st, next(x) = \tau) \Leftrightarrow \text{true}\)
• \(R_{cf}(h, st, \text{assume}(\sigma)) \Leftrightarrow R_{cf}(h, st, \text{assert}(\sigma)) \Leftrightarrow \text{true}\)
• \(R_{cf}(h, st, \text{nothing}) \Leftrightarrow \text{true}\)
• \(R_{cf}(h, st, \ell : \text{pause}) \Leftrightarrow (next(\ell) = st)\)
actions: assert and proof goals for verification, we also have to compute the guards of the
• R

if the condition

guard.

local variables, which is done by the data flow that will be defined in this

section.

The control flow refers to the values of input, local, and output variables to determine the locations that are be reached at the next point of
time. However, the control flow does not determine the values of output and
variables to determine the locations that are be reached at the next point of

The above recursive formulation of the control flow is almost half the way
towards the synthesis of hardware circuits. The differences are only that the
hardware synthesis will generate the same control flow as a conjunction of
transition equations of the location variables. Moreover, the hardware synthe-
sis maintains inputs for maintaining surrounding preemption statements.

• R

if

else

in terms of guarded actions,

• R

in terms of guarded actions,

• R

in terms of guarded actions,

4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

4.5.2 Computing Guarded Actions

In the previous section, we have defined the control flow of a statement S as
a finite state machine whose states are encoded by the location variables of
the statement. The control flow refers to the values of input, local, and output
variables to determine the locations that are be reached at the next point of
time. However, the control flow does not determine the values of output and
local variables, which is done by the data flow that will be defined in this
section.

We will define the data flow of a statement S in terms of guarded actions,
i.e., pairs (\(\gamma, C\)) where C is an atomic statement that is executed if and only
if the condition \(\gamma\) holds. Clearly, the only atomic statements C that modify
the values of variables are the assignments, so that it would be sufficient
to consider the assignments to define the data flow. However, to set up the
proof goals for verification, we also have to compute the guards of the assume
and assert statements. Hence, we have the following definition of guarded
actions:

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{if}(\sigma) & \ S_1 \ \text{else} \ S_2) \equiv (\mathcal{R}_{cf}(h, st \land h(\sigma), S_1) \land \\
& \mathcal{R}_{cf}(h, st \land \neg h(\sigma), S_2) \land \\
& (\text{in}(S_1) \rightarrow \neg \text{in}(S_2))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \{S_1; S_2\}) \equiv & \left( \mathcal{R}_{cf}(h, st, S_1) \land \\
& \mathcal{R}_{cf}(h, st \land \text{inst}_{h}(S_1) \lor \text{term}(S_1), S_2) \land \\
& (\text{in}(S_1) \rightarrow \neg \text{in}(S_2))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \{S_1 || S_2\}) \equiv & \left( \mathcal{R}_{cf}(h, st, S_1) \land \mathcal{R}_{cf}(h, st, S_2) \right)
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{do} \ S \ \text{while}(\sigma)) \equiv & \left( \mathcal{R}_{cf}(h, st \lor h(\sigma) \land \text{term}(S), S) \land \\
& (\sigma \land \text{term}(S) \rightarrow \neg \text{inst}_{h}(S))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \alpha \ x; \ S \ \}) \equiv & \mathcal{R}_{cf}(h', st, S), \ \text{where} \ h' := [h]^{h(\alpha)+1}_x
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{during} \ S_1 \ \text{do} \ S_2) \equiv & \mathcal{R}_{cf}(h, st, S)
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{[weak] abort} \ S \ \text{when}(\sigma)) \equiv & \left( \mathcal{R}_{cf}(st, S) \land \text{in}(S) \land \text{next}(\text{in}(S)) \rightarrow \neg \sigma) \lor \\
& \text{in}(S) \land \sigma \land \text{st} \land \text{inst}_{h}(S) \land \neg \text{next}(\text{in}(S)) \lor \\
& \text{in}(S) \land \sigma \land \text{st} \land \text{enter}_{h}(S) \lor \\
& \text{in}(S) \land \sigma \land \neg \text{st} \land \neg \text{next}(\text{in}(S))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{[weak] immediate abort} \ S \ \text{when}(\sigma)) \equiv & \left( \mathcal{R}_{cf}(st, S) \land (\text{in}(S) \rightarrow \neg \sigma) \lor \\
& (\text{in}(S) \land \sigma \land \text{stutter}(S))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{[weak] suspend} \ S \ \text{when}(\sigma)) \equiv & \left( \mathcal{R}_{cf}(st, S) \land (\text{in}(S) \rightarrow \neg \sigma) \lor \\
& (\text{in}(S) \land \sigma \land \text{stutter}(S))
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}_{cf}(h, st, \text{[weak] immediate suspend} \ S \ \text{when}(\sigma)) \equiv & \left( \mathcal{R}_{cf}(st, S) \land (\text{in}(S) \rightarrow \neg \sigma) \lor \\
& (\text{in}(S) \land \sigma \land \text{stutter}(S))
\end{align*}
\]
Definition 4.13 (Guarded Actions). A guarded action is a pair \((\gamma, C)\), where \(\gamma\) is a boolean valued expression called the guard, and \(C\) is either an immediate assignment \(x = \tau\), a delayed assignment \(\text{next}(x) = \tau\), an assumption \(\text{assume}(\sigma)\), or an assertion \(\text{assert}(\sigma)\).

The intuition behind a guarded action \((\gamma, C)\) is that the action \(C\) is executed if and only if the condition \(\gamma\) is satisfied. Guarded actions may be viewed as a single programming language like Unity [68] in that every guarded action runs as separate processes in parallel that observes the guard \(\gamma\) in each step, and executes \(C\) if the guard holds.

One might think that the set of guarded actions can be computed in a single pass as done for the control flow predicates of the previous section. In particular, one might wish to start with a precondition \(\varphi\) that holds when the (sub)statement is started and that is refined when the deeper nested substatements have to be started. However, we can not use the start condition as used in Definition 4.12 for this purpose, since the weak preemption statements make it necessary to distinguish between the start of the depth that allows the control flow to enter the statement and the start of the surface where the control flow will not enter the statement, but nevertheless some of its actions are enabled.

Hence, we might wish to start with the initial precondition \(st\) and to compute the guarded actions \(\text{Actions}(\varphi, S)\) of a statement \(S\) intuitively as follows (we neglect local declarations, since the definition below is already wrong for the mentioned statements):

- \(\text{Actions}(\varphi, x = \tau) := \{(\varphi, x = \tau)\}\)
- \(\text{Actions}(\varphi, \text{next}(x) = \tau) := \{(\varphi, \text{next}(x) = \tau)\}\)
- \(\text{Actions}(\varphi, \text{assume}(\sigma)) := \{(\varphi, \text{assume}(\sigma))\}\)
- \(\text{Actions}(\varphi, \text{assert}(\sigma)) := \{(\varphi, \text{assert}(\sigma))\}\)
- \(\text{Actions}(\varphi, \text{nothing}) := \{\}\)
- \(\text{Actions}(\varphi, \ell: \text{pause}) := \{\}\)
- \(\text{Actions}(\varphi, \text{if}(\sigma) S_1 \text{ else } S_2) := \text{Actions}(\varphi \land \sigma, S_1) \cup \text{Actions}(\varphi \land \neg \sigma, S_2)\)
- \(\text{Actions}(\varphi, \{S_1; S_2\}) := \text{Actions}(\varphi, S_1) \cup \text{Actions}(\varphi \land \text{inst}(S_1) \lor \text{term}(S_1), S_2)\)
- \(\text{Actions}(\varphi, \{S_1 \parallel S_2\}) := \text{Actions}(\varphi, S_1) \cup \text{Actions}(\varphi, S_2)\)
- \(\text{Actions}(\varphi, \text{do } S \text{ while}(\sigma)) := \text{Actions}(\varphi \lor \sigma \land \text{term}(S), S)\)
- \(\text{Actions}(\varphi, \text{during } S_1 \text{ do } S_2) := \text{Actions}(\text{in}(S_1), S_2) \cup \text{Actions}(\varphi, S_1)\)
- \(\text{Actions}(\varphi, \text{abort } S \text{ when}(\sigma)) := \{(\gamma \land (\text{in}(S) \rightarrow \neg \sigma), C) \mid (\gamma, C) \in \text{Actions}(\varphi, S)\}\)

However, the above definition is already wrong for the definition of the strong abortion statement. However, before showing its deficiencies, let us advocate in favor of the definitions of the strong abortion. The idea of the above definition is to compute the guarded actions \(\text{Actions}(\varphi, S)\) as if no abortion context
would exist. Then, we have to disable the actions whenever the abortion condition $\sigma$ is false. However, a simple conjunction $\gamma \land \sigma$ is not correct, since there are actions $C$ that are executed in $S$ before the control flow enters $S$. As the abortion is not an immediate one, we have to distinguish between those actions that are executed in the depth of the body $S$ and those that are executed in the surface, i.e., when entering $S$ from outside. For this reason, the conjunct $(S) \rightarrow \neg \sigma$ is added to the guards which disables the actions when the control flow is inside and $\sigma$ holds.

However, this definition is wrong which is demonstrated by the following example:

```plaintext
module RedoAbort(event i,&a,&b) {
    do
        abort
        emit a;
        $\ell$:pause;
        emit b;
        when(i)
        while(true)
    }
}
```

Assume that the control flow rests currently at the `pause` statement and that $i$ holds. Then, $b$ is not emitted (due to the strong abortion) and the strong abortion statement instantaneously terminates. Therefore, the loop body is re-entered, and the strong abortion statement is therefore also re-entered. While entering the abortion statement, the emission of $a$ takes place, since the abortion is not an immediate one, and then the control stops again at the `pause` statement.

However, according to the above definition of $\text{Actions}(st,S)$, we would obtain the two guarded actions $((st \lor \ell) \land (\ell \rightarrow \neg i), \text{emit } a)$ and $((\ell \land (\ell \rightarrow \neg i), \text{emit } b)$. Hence, if $\ell$ holds, the guarded actions are simplified to $(-i, \text{emit } a)$ and $(-i, \text{emit } b)$ which is wrong for the emission of $a$. The correct guarded actions should be rather $(st \lor \ell, \text{emit } a)$ and $(\ell \land \neg i, \text{emit } b)$.

So, what went wrong? The problem is that we have to distinguish for delayed preemption statements between those actions that are executed by entering their body statements and the remaining actions that are executed in the depth of the body statement. If we could present these actions in two different sets, then we can simply add the conjunct $\neg \sigma$ to the guards of the actions of the depth, while the guards of the actions of the surface remain unchanged. For this reason, we have to separately compute the actions of the surface $\text{ActSurf}_h(\phi,S)$ and the actions of the depth $\text{ActDepth}(S)$.

Another reason for distinguishing between actions of the surface and the depth is that we have to distinguish between different scopes of locally declared variables. As we have explained in Section 3.2, the local declaration of a variable can be left and re-entered several times within a macro step, and we have to distinguish the different reincarnations of the local variable. The
computation of the surface actions refers to one of these reincarnations, so that we have
to add an incarnation level function ℏ as a further argument that tells us the incarnation
levels of the variables. Using ℏ, we can appropriately endow the variable occurrences
with their incarnation indices, so that we can safely distinguish between the different scopes,
when guarded actions of different surfaces are merged later on.

\[
\text{function } \text{DisableDelayed}(L, \varphi, G) \\
G' := \{\}; \\
\text{for every } (\gamma, C) \text{ do} \\
\quad \text{case } C \text{ of} \\
\quad \quad \text{next}(x) = \tau : \text{ if } x \in L \text{ then } G' := G' \cup \{(\gamma \land \neg \varphi, \text{next}(x) = \tau)\}; \quad \text{else } G' := G' \cup \{(\gamma, \text{next}(x) = \tau)\}; \\
\quad \text{end} \\
\text{end}; \\
\text{return } G'; \\
\text{end}
\]

Fig. 4.9. Disabling Delayed Local Assignments

Moreover, we have to make sure that no delayed action on \(x\) is executed in case that \(S\) is instantaneously executed. This is done by the function \(\text{DisableDelayed}(L, \varphi, G)\) shown in Figure 4.9 that replaces each guard \((\gamma, \text{next}(x) = \tau) \in G\) of a delayed assignment to a variable \(x \in L\) by \((\gamma \land \neg \varphi, \text{next}(x) = \tau) \in G\). Hence, the delayed assignments on \(x\) are disabled whenever \(\varphi\) holds.

The definition of the guarded surface actions is then as follows, where \(\text{LocVar}(S)\) computes the set of locally declared variables in \(S\):

**Definition 4.14 (Guarded Actions of the Surface).** Given a Quartz statement \(S\), an incarnation level function \(h\), and a precondition \(\varphi\), the following recursively defined function computes the set of guarded surface actions:

- \(\text{ActSurf}_h(\varphi, x = \tau) := \{(\varphi, h(x) = h(\tau))\}\)
- \(\text{ActSurf}_h(\varphi, \text{next}(x) = \tau) := \{(\varphi, \text{next}(x) = h(\tau))\}\)
- \(\text{ActSurf}_h(\varphi, \text{assume}(\sigma)) := \{(\varphi, \text{assume}(h(\sigma)))\}\)
- \(\text{ActSurf}_h(\varphi, \text{assert}(\sigma)) := \{(\varphi, \text{assert}(h(\sigma)))\}\)
- \(\text{ActSurf}_h(\varphi, \text{nothing}) := \{\}\)
- \(\text{ActSurf}_h(\varphi, \ell: \text{pause}) := \{\}\)
- \(\text{ActSurf}_h(\varphi, \text{if}(\sigma) \ S_1 \text{ else } S_2) := \text{ActSurf}_h(\varphi \land h(\sigma), S_1) \cup \text{ActSurf}_h(\varphi \land \neg h(\sigma), S_2)\)
- \(\text{ActSurf}_h(\varphi, \{S_1; S_2\}) := \text{ActSurf}_h(\varphi, S_1) \cup \text{ActSurf}_h(\varphi \land \text{inst}_h(S_1), S_2)\)
- \(\text{ActSurf}_h(\varphi, \{S_1 \parallel S_2\}) := \text{ActSurf}_h(\varphi, S_1) \cup \text{ActSurf}_h(\varphi, S_2)\)
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates 119

- ActSurfₜ(φ, do S while(σ)) := ActSurfₜ(φ, S)
- ActSurfₜ(φ, {α x; S}) := DisableDelayed(\{x\}, instₜ(S), ActSurfₜ(φ, S))
- ActSurfₜ(φ, during S₁ do S₂) := ActSurfₜ(φ, S₁)
- ActSurfₜ(φ, [weak] abort S when(σ)) := ActSurfₜ(φ, S)
- ActSurfₜ(φ, [weak] immediate abort S when(σ)) := DisableDelayed(LocVar(S), σ, ActSurfₜ(φ, S))
- ActSurfₜ(φ, immediate abort S when(σ)) := ActSurfₜ(φ, S)
- ActSurfₜ(φ, [weak] suspend S when(σ)) := ActSurfₜ(φ, S)
- ActSurfₜ(φ, immediate suspend S when(σ)) := ActSurfₜ(φ, S)

The above definition for the basic statements should be clear: we take the so-far computed precondition φ as the guard for the atomic action that has to be renamed according to the current scopes given by the incarnation level function ℏ. Note that in case of delayed assignments only the right hand side τ must be renamed, while the left hand side always refers to the depth. In general, we have to apply ℏ to all expressions that are to be evaluated in the surface.

For the conditional statement, we simply add ℏ(σ) or its negation to the precondition to start the corresponding substatement. According to the semantics of a sequence, we first execute S₁. If the execution of S₁ is instantaneous, then we also execute S₂ in the same macro step. Hence, the precondition for the surface actions of S₂ is \(\phi \land \text{inst}_h(S_1)\). The preconditions of the substatements of a parallel statement are simple the precondition of the parallel statement.

The do-while loop always enters its body statement, and therefore, we forward the precondition of the loop to the computation of the surface actions of the body statement. Note that the body statement is also entered when the body statement terminates, but that is another surface that will be computed with a potentially different incarnation level function.

The surface actions of a local declaration \(\{\alpha x; S\}\) are simply the surface actions of its body S. Note that we do not increment the incarnation level function ℏ here since the correct incarnation levels must be provided for the computation of the surface actions. This is done by the computation of the depth actions defined below. Note also that we disable all delayed assignments to the local variable x in case that S is instantaneous.

At starting time, the during statement only executes the surface of its first substatement S₁. Therefore, the surface actions are only computed of S₁.

As delayed preemptions are ignored at starting time of a delayed preemption statement, we can ignore them for the computation of the surface actions. The same holds for the weak-immediate variants, since even if the preemption takes place, the actions are still retained due to the weak preemption. However, there is the exception that we have to disable delayed actions on local variables when their scope is left by an abortion (regardless whether this
is weak or strong). Finally, the strong preemption statements are handled by adding the conjunct $\neg h(\sigma)$ to the precondition $\varphi$.

It is easily seen by induction on the above definition that the following statements are valid:

- The guard $\gamma$ of a guarded surface action $(\gamma, C)$ only refers to the current values of the input, output, and local variables, where for the latter the incarnation levels as provided by $h$ are chosen.
- The runtime to compute $\text{ActSurf}_h(\varphi, S)$ is $O(|S|)$, since the recursive calls describe a single pass over the syntax tree of $S$. To this end, we preliminarily assume that the control flow predicates are already available.
- Every guard $\gamma$ of a guarded action $(\gamma, C) \in \text{ActSurf}_h(\varphi, S)$ implies $\varphi$.

It remains to compute the remaining actions which are executed in the depth of the statement $S$. These actions are enabled by the resumption of the control flow that already rests somewhere in the statement. For this reason, we do not need a precondition as argument since the precondition is the active current control flow location itself.

For the same reason, we do not need an incarnation level function $h$, since we start the resumption of the execution with the incarnation level function $h_0$ that maps all variables to 0. As the computation of the depth actions will invoke further calls to surface actions, the computation of the depth actions will also compute an incarnation level function that can be forwarded to the computation of these surface actions.
Definition 4.15 (Guarded Actions of the Depth). Given a Quartz statement $S$, the following recursively defined function $\text{ActDepth}(S)$ computes a triple $(h, \Xi, A)$, where $h$ is the incarnation level function that points to the scopes after execution of the depth of $S$, $\Xi$ holds the starting conditions of the loop surfaces contained in the depth of $S$, and $A$ is the set of the guarded actions of the depth:

- $\text{ActDepth}(S) := (h_0, \emptyset, \emptyset)$ for all atomic statements $S$
- $\text{ActDepth}(\text{if} (\sigma) \ S_1 \ \text{else} \ S_2) := (\text{Max}(h_1, h_2), \Xi_1 \cup \Xi_2, A_1 \cup A_2)$, where $(h_1, \Xi_1, A_1) := \text{ActDepth}(S_1)$ and $(h_2, \Xi_2, A_2) := \text{ActDepth}(S_2)$
- $\text{ActDepth}(\{S_1; S_2\}) := (\text{Max}(h_1, h_2), \Xi_1 \cup \Xi_2, A_1 \cup A_2)$, where $(h_1, \Xi_1, A_1) := \text{ActDepth}(S_1)$ and $(h_2, \Xi_2, A_2) := \text{ActDepth}(S_2)$
- $\text{ActDepth}(\text{do } S \ \text{while} (\sigma)) := (h_2, \Xi \cup \Xi, A_1 \cup A_2)$, where $(h_1, \Xi, A_1) := \text{ActDepth}(S)$
- $\text{ActDepth}(\text{during } S_1 \ \text{do} \ S_2) := (h_1, \Xi_1, A_1 \cup A_2)$, where $(h_1, \Xi, A, 1) := \text{ActDepth}(S_1)$
- $\text{ActDepth}(\text{weak [immediate] abort } S \ \text{when} (\sigma)) := \text{DisableDelayed}(\text{LocVar}(S), \sigma, \text{ActDepth}(S))$
- $\text{ActDepth}(\text{[immediate] abort } S \ \text{when} (\sigma)) := (h_1, \Xi, A_2)$, where $(h_1, \Xi, A_1) := \text{ActDepth}(S)$
- $\text{ActDepth}(\text{weak suspend } S \ \text{when} (\sigma)) := \text{ActDepth}(S)$
- $\text{ActDepth}(\text{suspend } S \ \text{when} (\sigma)) := (h_1, \Xi, A_1)$, where $(h_1, \Xi, A) := \text{ActDepth}(S)$
- $\text{ActDepth}(\text{do weak immediate suspend } S \ \text{when} (\sigma)) := (h_1, \Xi, A_1 \cup A_2)$, where $(h_1, \Xi_1, A_1) := \text{ActDepth}(S)$ and $A_2 := \text{ActSurf}_{h_0}(\ell, S)$
- $\text{ActDepth}(\text{do immediate suspend } S \ \text{when} (\sigma)) := (h_1, \Xi, A_1 \cup A_2)$, where $(h_1, \Xi, A_1) := \text{ActDepth}(S)$ and $A_2 := \{((\gamma, C) \in \text{ActSurf}_{h_0}(\ell, S)) | (h_1, \Xi, A_1) := \text{ActDepth}(S) \}$

Statements without control flow locations do not have depth actions. This explains the definitions of the atomic statements except for pause, which is however obvious.
As the control flow can rest in one of the branches of an if-statement, it can be resumed from any of these branches. We therefore simply take the ‘union’ of the two computations of the depth actions. For if-statements, sequences, and parallel statements, we merge again the obtained incarnation level functions. Note once more that this is not problematic, since the incrementations of the incarnation level function done in the separate substatements refer to the local variables of those substatements, and therefore, to different local variables.

Similarly, the control flow can rest in either one of the substatements \( S_1 \) or \( S_2 \) of a sequence \( \{S_1 ; S_2\} \), and hence, we can resume it from either \( S_1 \) or \( S_2 \). If the control flow is resumed from somewhere inside \( S_1 \), and \( S_1 \) terminates, then also the surface actions of \( S_2 \) are executed in the depth of the sequence.

Note that the computation of the depth of a sequence \( \{S_1 ; S_2\} \) leads to the computation of the surface actions of \( S_2 \). Even though this is a copy of the surface of \( S_2 \) that occurs in the depth of \( \{S_1 ; S_2\} \) (the other copy occurs in the surface of \( \{S_1 ; S_2\} \)), there is no need to increase the incarnation levels in this copy. It is the case that this copy of the surface can be executed at the same macro step as the copy in the surface. However, then a surrounding loop is required and the incrementation of the incarnation levels is done there. We consider this discussion in more detail in Section 5.1.

The parallel statement \( \{S_1 \parallel S_2\} \) can hold the control flow in each of its substatements, and in contrast to sequences and if-statements, the control flow may even rest in both substatements at the same time. The computation of the depth actions is simply the ‘union’ of the depth actions of the two substatements.

The most interesting case is the computation of the depth actions of loops, since we have to handle the incarnation problem there: Clearly, we first compute the depth actions of the loop’s body statement. However, similar to sequences, it may be the case (namely when the loop body terminates and the loop condition holds) that the body statement is re-entered. If the loop body contains a local variable declaration, then the surface of the loop body may contain a part of the local declaration while the depth contains another one. Here, we have to distinguish the two different scopes to avoid confusion with the schizophrenic local declarations. For this reason, the incarnation levels of all variables locally declared in \( S \) are incremented by one before computing the surface actions.

Note that the only case, where the incarnation level function is modified is in the computation of loops. The computation done there is moreover pessimistic, since the worst case is considered where the scopes of all local variables declared in \( S \) are suspected to be re-entered. Heuristics could be used here to reduce the number of reincarnations by limiting the incrementation of the incarnation levels to those variables whose scopes can really be re-entered by the loop.

Moreover, loops are the only statements where the additional result \( \Xi \) is modified. In this set, we store triples of the form \((\mathcal{L}, h, \varphi)\) for every surface that
has been generated by a loop. This information is required for the construction of the data flow transition relation as described in the next section. To this end, it is necessary to know the preconditions $\varphi$ of the surfaces generated by loops that are contained in the depth together with the reincarnated variables (that can be derived from $L$ and $h$).

The depth actions of a local declaration are simply the depth actions of the body statement. We do not have to worry about incarnation levels here, since we start in the depth of the local declaration. However, we have to disable all delayed actions that would otherwise be executed at termination time of the local declaration. This is again done by the function call $\text{DisableDelayed}(x, \text{term}(S), A)$ for the function defined in Figure 4.9.

The second substatement of the during $S_1$ do $S_2$ statement must always be instantaneous. Therefore, the depth actions are only obtained by resuming the control flow from all possible locations inside $S_1$. However, before resuming the execution in $S_1$, the semantics of the during statement is to execute the instantaneous statement $S_2$.

Weak preemption statements can be ignored for the computation of the depth actions, since even if the abortion takes place, all actions remain enabled due to the weak preemption. However, similar to the surface actions, we have to disable all delayed actions on local variables since their scopes are left due to the abortion (regardless whether it is weak or strong).

For a strong abortion, we add a conjunct $\neg \sigma$ to the guards of all actions to disable them in case $\sigma$ holds. Note that we do not have to care about incarnation levels in $\sigma$ since $\sigma$ is here evaluated in the depth: Even if the body statement $S$ enters some scopes of local declarations, these variables can not occur in $\sigma$, and local variables that are declared around the preemption statement can not be reincarnated by the resumption of the abortion’s body statement. The same explanations hold for the strong suspension statement.

The immediate versions of the suspension are a bit more difficult since the control flow could have been caught either inside the body statement or on the additional location of the immediate suspension. A weak immediate suspension can therefore execute both the surface and depth actions of its body statement. The strong variant must furthermore disable the guards when the suspension condition $\sigma$ holds.

Similar to the computation of the surface actions, we can easily observe the following:

- The guard $\gamma$ of a guarded depth action $(\gamma, C)$ only refers to the current values of the input, output, and local variables, where for the latter the right incarnation levels are chosen.
- The runtime to compute $\text{ActDepth}(S)$ is $O(|S|^2)$, since the recursive calls describe a single pass over the syntax tree of $S$, but in each call, it is possible to invoke a computation of the surface which requires time $O(|S|)$. Again, we preliminarily assume that the control flow predicates are already available.
In the previous section, we have shown how the guarded actions of a statement \( S \) can be computed in terms of its surface actions \( \text{Act}_{\text{surf}}(\text{st}, S) \) and its depth actions \( \text{Act}_{\text{depth}}(S) \). For the initial computation, we use the start signal or location \( \text{st} \) as precondition of the surfaces and the zero incarnation level function \( h_0 \) that maps all variables to incarnation level 0. The guarded actions of a statement \( S \) are therefore \( \text{Act}_{\text{surf}}(h_0(\text{st}, S)) \cup \text{Act}_{\text{depth}}(S) \).

The guarded actions are the essential information that we need to define the data flow of the program, which is explained in this section. The meaning of a guarded action \((\gamma, C)\) is that whenever \(\gamma\) holds, then \(C\) is executed. Executing an immediate assignment \(x = \tau\) has the effect that immediately the equation \(x = \tau\) holds, while the execution of a delayed assignment means that the value of \(x\) at the next point of time is equal to the current value of \(\tau\). Assumptions and assertions do not contribute to the data flow. Instead, they are used for the construction of specifications as described in the next subsection.

For the following, we therefore only consider the guarded actions that are assignments. Moreover, we partition them according the left hand side variables (expressions). For example, assume that for a variable \(x\), we have obtained the guarded actions \((\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p)\) with immediate assignments, and the guarded actions \((\chi_1, \text{next}(x) = \pi_1), \ldots, (\chi_q, \text{next}(x) = \pi_q)\) with delayed assignments. Then, the following formulas must hold at every point of time:

\[
\left( \bigwedge_{i=1}^{p} (\gamma_i \rightarrow (x = \tau_i)) \right) \land \left( \bigwedge_{i=1}^{q} (\chi_i \rightarrow (\text{next}(x) = \pi_i)) \right)
\]

Note that this formula can become false if there are write conflicts on the variable \(x\). We can essentially distinguish three kinds of write conflicts:

- If two immediate assignments \(x = \tau_i\) and \(x = \tau_j\) are enabled at the same time, then the formula is false unless we have \(\tau_i = \tau_j\) at that point of time.
- If two delayed assignments \(\text{next}(x) = \pi_i\) and \(\text{next}(x) = \pi_j\) are enabled at the same time, then the formula is false unless we have \(\pi_i = \pi_j\) at that point of time.
- If a delayed assignment \(\text{next}(x) = \pi_i\) is enabled at the current point of time, and an immediate assignment \(x = \tau_i\) is enabled at the next point of time, then the formula is false at the next point of time unless \(\pi_i\) of the current point of time is the same value as \(\tau_i\) at the next point of time.

Hence, write conflicts can reduce the entire data flow formula to false. Checking for write conflicts is however not simple for the data flow transition relation that is developed in this section. We consider the analysis of write conflicts in more detail in Chapter 8.

The above formula for the data flow of the variable \(x\) is not yet complete. It only determines the value of \(x\) at a point of time \(t\) when an immediate
assignment is executed at time $t$ or a delayed assignment has been executed at time $t-1$. However, if neither is the case, then we must also determine a value for $x$ which is done by the reaction-to-absence that has already been described in Figure 4.8. In this case, $x$ takes its default value $\text{Default}(x)$ which is defined for memorized variables as the previous value of $x$, and for event variables as the default value of the type of $x$ (which is 0 for numbers and $\text{false}$ for booleans). A precise definition of the data flow is given in Figure 4.10.

Figure 4.10 considers thereby the more general case where $x$ is a locally declared variable that has some reincarnations. The case of global output variables is a special case of local variables where no reincarnations have to be considered.

To describe this formally, assume we have computed for a local variable $x$ and its $d$ reincarnations $x_1, \ldots, x_d$ the following guarded actions (note that there are no delayed actions on the reincarnations), where $x_d$ is assumed to be the outermost reincarnation of $x$:

- $(\chi_1, \text{next}(x) = \pi_1)$
- $(\gamma_1, x = \tau_1)$
- $(\gamma_{1,1}, x_1 = \tau_{1,1})$
- $\ldots$
- $(\gamma_{1,p}, x_1 = \tau_{1,p})$
- $\ldots$
- $(\gamma_{d,1}, x_d = \tau_{d,1})$
- $\ldots$
- $(\gamma_{d,p_d}, x_d = \tau_{d,p_d})$

Reincarnations $x_i$ do always have event storage mode, since they only exist for one point of time. Note further that we have no delayed actions on reincarnated variables, since the renamings due to the incarnation level function $\mathcal{h}$ have not been applied on the left hand sides of delayed assignments. For this reason, the transition relation of a reincarnated variable $x_i$ is defined as the invariant $\text{Invar}_{x_i}$ in Figure 4.10 that has to hold at every point of time. This invariant simply states that whenever the trigger condition $\gamma_{i,j}$ holds, then the corresponding equation $x_i = \tau_{i,j}$ must also hold. If no trigger condition $\gamma_{i,j}$ holds, then it is required that $x_i$ equals to the default value $\text{Default}(x)$.

The initial condition $\text{Init}_x$ of the variable $x$ is constructed analogously to $\text{Invar}_{x_i}$ which indicates that the semantics of reincarnated variables corresponds with an initialization of a variable. The explanations for the construction of $\text{Init}_x$ are the same as for $\text{Invar}_{x_i}$.

The transition relation of $x$ is more difficult: We have to distinguish between event and memorized variables $x$ for the reaction to absence. In any case, the equation $x = \tau_j$ must hold if the trigger condition $\gamma_j$ holds. Moreover, if the trigger condition $\chi_j$ of a delayed assignment $\text{next}(x) = \pi_j$ holds, then $x$ must have the value $\pi_j$ at the next point of time (note that $\pi_j$ is evaluated with the current variables to determine the value of $x$ for the next point of time).

If neither a trigger condition $\gamma_j$ of an immediate assignment nor a trigger condition of a delayed assignment $\chi_j$ holds, then we have to implement
Computed Guarded Commands of \( x \) and its Reincarnations \( x_1, \ldots, x_d \):
\[
\begin{align*}
(\chi_1, \text{next}(x) = \pi_1), \ldots, (\chi_q, \text{next}(x) = \pi_q) \\
(\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p), \\
(\gamma_{1,1}, x_1 = \tau_{1,1}), \ldots, (\gamma_{p,1}, x_1 = \tau_{p,1}), \\
\vdots & \\
(\gamma_{1,d}, x_d = \tau_{1,d}), \ldots, (\gamma_{p,d}, x_d = \tau_{p,d})
\end{align*}
\]

Transition Relation of Reincarnated Variable \( x_i \):
\[
\text{Invar}_{x_i} := \left( \bigwedge_{j=1}^{p} (\gamma_{i,j} \rightarrow x_i = \tau_{i,j}) \right) \land \left( \bigwedge_{j=1}^{q} (\chi_{j} \rightarrow \text{next}(x_i) = \pi_j) \right)
\]

Initial Condition of Event/Memorized Variable \( x \):
\[
\text{Init}_x := \left( \bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \right) \land \left( \bigwedge_{j=1}^{q} (\chi_j \rightarrow x = \tau_j) \right)
\]

Transition Relation of Variable \( x \):
\[
\text{Trans}_x := \left( \bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \right) \land \left( \bigwedge_{j=1}^{q} (\chi_j \rightarrow \text{next}(x) = \pi_j) \right) \land \\
\left( \text{next} \left( \bigwedge_{j=1}^{p} (\neg \gamma_j) \land \bigwedge_{j=1}^{q} (\neg \chi_j) \right) \rightarrow \text{next}(x) = \text{Initialize}(x) \right)
\]

where the expression \( \text{Initialize}(x) \) is defined as follows:
\[
\text{Initialize}(x) := \left\{ \begin{array}{ll}
\text{Default}(x) & : \text{if } x \text{ is an event variable} \\
\text{case } \begin{array}{l}
g_0 : x_d; \\
\vdots \\
g_0 : x_1; \\
\text{else } x
\end{array} & : \text{if } x \text{ is a memorized variable}
\end{array} \right.
\]

Fig. 4.10. Transition Relation to Define the Data Flow of Local Event/Memorized Variables
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

the reaction-to-absence: The value of an event variable \( x \) is the default value \( \text{Default}(x) \) associated with the type of \( x \).

The reaction-to-absence for memorized variables is more difficult. Clearly, if an immediate or delayed assignment is executed, we have the same effect as for event variables, and therefore the first two conjuncts of \( \text{Trans}_x \) are the same for event and memorized variables. The reaction-to-absence is however more difficult for memorized variables, since there may be an interference of reincarnated variables and the original memorized variable \( x \): It may be the case that there is neither a delayed assignment to \( x \) in the surface of the local declaration nor an immediate assignment right after this point of time. In this case, \( x \) has to capture the value of that reincarnated variable \( x_i \) that corresponds with the surface that has been executed most recently.

Hence, we have to determine the 'most recently executed surface' in order to capture the value of the most recently reincarnated variable. This is the reason why the preconditions of each surface that has been generated by a loop has been stored in the variable \( \Xi \) during the computation of the depth actions. Using these preconditions, we are able to check one after the other (in the ordering of their nestings) to determine the most recently activated surface. Note that this selection is done dynamically and the formulas we generate therefore have to cover all possible cases.

Hence, assume that \( go_i \) is the precondition of a loop surface where the local variable \( x \) has been reincarnated to \( x_i \) (hence, \( \bigvee_{j=1}^{p_i} \gamma_{i,j} \rightarrow go_i \) must be valid, but the converse may not hold). The transition relation for \( x \) is then determined by the expression \( \text{Initialize}(x) \) in Figure 4.10. With the case construct, we check the preconditions \( go_i \) one after the other. Note that if \( go_i \) is the first one that holds, then all \( go_j \) with \( j < i \) do also hold, which means that \( go_i \) refers to the outermost surface whose reincarnated variable has to be referred to. Finally, if no precondition \( go_i \) holds, then the control flow moved from somewhere inside the scope of \( x \) to somewhere inside the scope of \( x \) without leaving the scope in between. Hence, we simply store the previous value of \( x \), which is the default case of the case construct in the expression \( \text{Initialize}(x) \).

Again, it is possible that write conflicts can appear in the transition relations \( \text{Trans}_x \) and \( \text{Trans}_X \). Note that it is no problem if several surfaces are executed and if the local variable is assigned different values in each of these surfaces. We consider the issue of write conflicts in Chapter 8 in more detail.

4.5.4 Haltset Encoding of Symbolic SOS Rules

In the previous sections, we have defined the semantics of Quartz statements by means of the SOS transition and reaction rules that are supplemented by the reaction-to-absence as given in Sections 4.2-4.4. The symbolic definition of the same semantics has been given by the control flow predicates of Section 4.5.1 and the transition relation of the data flow as defined in the previous section.
It is easily seen that all control flow predicates can be computed in a single pass over the syntax tree of the statement which can be done in time $O(|S|)$. To this end, we remark that it is necessary to share common subexpressions, since otherwise an exponential blow-up would be obtained. It is moreover possible to compute the surface actions in the same pass which can still be done in time $O(|S|)$. However, adding the computation of the depth actions will then require time $O(|S|^2)$, since the depth triggers the computation of further surface actions taking time $O(|S|)$. We consider the aspect of computing all control flow predicates and all guarded actions in the next chapter, where we refine the control flow once more: We will see that the control flow can always be generated as a system of transition equations where each location variable has a single initialization and transition equation.

In the remainder of this section, we consider the relationship between the SOS transition rules and the control flow predicates. To this end, we have to derive a variant of the SOS transition rules that is based on an encoding of the residual statements by so-called haltsets [40, 42], which are sets of location variables. Of course, the meaning of a haltset $H$ is that the locations contained in $H$ are currently active.

Not every haltset is a consistent one: The control flow of sequences and if-statements leads to the restriction that at most one of the substatements may be active at the same point of time. Therefore, not every subset of $\text{lbls}(S)$ corresponds with a residual statement of $S$, however, every consistent subset does. The following definition determines the encoding of residual statements by a consistent haltset $H$:

**Definition 4.16 (Residual Statement of Haltset).** The following recursively defined function computes for every statement $S$, every boolean flag $s$, and every consistent set $H \subseteq \text{lbls}(S)$ the corresponding residual statement $\text{Residual}_{\text{H}}(S)$:

- $\text{Residual}_{\text{true}}(S) = S$
- $\text{Residual}_{\text{false}}(S) = \text{nothing}$, otherwise
  - $\text{Residual}_{\text{H}}(x = \tau) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\text{next}(x) = \tau) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\text{assume}(\sigma)) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\text{assert}(\sigma)) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\text{nothing}) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\ell: \text{pause}) := \text{nothing}$
  - $\text{Residual}_{\text{H}}(\text{if}(\sigma) S_1 \text{ else } S_2)$
    $$\text{:= } \begin{cases} \text{Residual}_{\text{H}}(S_1) : \text{ if } H \subseteq \text{lbls}(S_1) \\ \text{Residual}_{\text{H}}(S_2) : \text{ if } H \subseteq \text{lbls}(S_2) \end{cases}$$
  - $\text{Residual}_{\text{H}}(\{S_1; S_2\})$
    $$\text{:= } \begin{cases} \text{Residual}_{\text{H}}(S_1) ; S_2 : \text{ if } H \subseteq \text{lbls}(S_1) \\ \text{Residual}_{\text{H}}(S_2) : \text{ if } H \subseteq \text{lbls}(S_2) \end{cases}$$
4.5 Symbolic SOS Transition Rules: The Control Flow Predicates

- Residual$^S_H$\{\{S_1 \parallel S_2\}\}

\[
\begin{align*}
\text{Residual}^S_H(S_1) & : if \ H \subseteq \text{lbs}(S_1) \\
\text{Residual}^S_H(S_2) & : if \ H \subseteq \text{lbs}(S_2) \\
\text{Residual}^S_{H \cap \text{lbs}(S_1)}(S_1) \parallel \text{Residual}^S_{H \cap \text{lbs}(S_2)}(S_2) & : otherwise
\end{align*}
\]

- Residual$^S_H$(do $S$ while($\sigma$)) := \{Residual$^S_H(S)$; while($\sigma$) $S$\}
- Residual$^S_H$(while($\sigma$) $S$) := \{Residual$^S_H(S)$; while($\sigma$) $S$\}
- Residual$^S_H$(do $x; S$) := Residual$^S_H(S)$
- Residual$^S_H$(during $S_1$ do $S_2$) := \{ $S_2$; during Residual$^S_H(S_1)$ do $S_2$ \}
- Residual$^S_H$(suspend $S$ when($\sigma$))

\[
\begin{align*}
\text{Residual}^S_H(\ell:\text{immediate suspend } H(S)) & : if \ H = \{\ell\} \\
\text{Residual}^S_H(\ell:\text{weak immediate suspend } H(S)) & : otherwise
\end{align*}
\]

- Residual$^S_H$(weak suspend $S$ when($\sigma$))
- Residual$^S_H$(weak immediate suspend $S$ when($\sigma$))

\[
\begin{align*}
\text{Residual}^S_H(\ell:\text{weak immediate suspend } H(S)) & : if \ H = \{\ell\} \\
\text{Residual}^S_H(\ell:\text{weak immediate abort } H(S)) & : otherwise
\end{align*}
\]

- Residual$^S_H$(suspend $S$ when($\sigma$))
- Residual$^S_H$(weak abort $S$ when($\sigma$))
- Residual$^S_H$(weak immediate abort $S$ when($\sigma$))

Due to the consistency of $H$, the case distinctions made for sequences and if-statements are complete. For the empty haltset $H = \{\}$, we have to distinguish between the situation where the control flow did not yet enter the statement, and the situation where the control flow may have already left it. This distinction was also the reason why we had to introduce a start signal or location for the symbolic definition of the control flow.

Using the haltset encoding of residuals, we can recast the SOS transition rules of a statement $S$ in terms of its haltsets. To this end, we would replace the statements $S_1$ and $S_2$ of a SOS transition rule $\langle E, h, S_1 \rangle \rightarrow \langle h', S_2, D, t \rangle$ by haltsets that encode it. Hence, we would write $S : \langle E, h, H_1 \rangle \rightarrow \langle h', H_2, D, t \rangle$ for $S_1 = \text{Residual}^\text{true}_H(S)$ and $S_2 = \text{Residual}^\text{false}_H(S)$. Using this haltset encoding, it has been formally proved in [224] that the following relationship holds between the control flow predicates and the SOS transition rules:

- instantaneous transitions: $H_1 = \{\}$ and $H_2 = \{\}$
- entering transitions: $H_1 = \{\}$ and $H_2 \neq \{\}$
- internal transitions: $H_1 \neq \{\}$ and $H_2 \neq \{\}$
- terminating transitions: $H_1 \neq \{\}$ and $H_2 = \{\}$
It has been proved in [224] that using $H_1$ and $H_2$ as variable assignments for the current and next values of the location variables yields satisfying assignments for the control flow predicates under the above cases. In a similar way, the surface and depth actions can be related to the SOS transition rules: The surface actions of $S$ are those that appear on the SOS transition rules from haltset $H_1 = \emptyset$, while the depth actions are obtained as labels when starting from the haltset $H_1 \neq \emptyset$. Program expressions $\sigma$ that lead to different SOS rules have to handled symbolically to this end, as described in [224].

### 4.6 Symbolic SOS Reaction Rules

The control flow predicates of Section 4.5 are a symbolic version of the SOS transition rules. The symbolic version of the SOS transition rules has the advantage that we can encode the system’s behavior for all possible inputs and reachable states in remarkably small formulas. This can be used for the analysis of the program’s behavior as done in formal verification (see Chapter 8) as well as for compilers as we will discuss in the next chapter.

For the semantics of Quartz, we have however also defined the SOS reaction rules that are used to incrementally determine the current values of the outputs by a fixpoint iteration. To be able to analyze the full semantics of Quartz in terms of symbolic descriptions, we therefore also need a symbolic form of the SOS reaction rules. We will develop a symbolic version of the SOS reaction rules in this section.

#### 4.6.1 Causality Analysis with Higher Data Types

In Figure 4.11 on Page 134, we have shown how the transition for the data flow of a local/output variable $x$ is determined. However, this transition relation is given at the granularity of macro steps, and does therefore not respect the SOS reaction rules that determine the constructive reaction of a program.

To analyze the constructiveness of a program, we have to formulate the data flow at the finer level of micro steps. To this end, we have to model the progress of information since the values of the local/output variables are determined step by step in terms of micro steps. To this end, we formally have to add an explicit unknown value $\bot$ to every data type, so that we can express that the value of a variable is not yet known at a particular micro step. These ‘ternary’ environments have already been used in Section 4.1 to describe the constructive operational semantics of Quartz by means of the SOS reaction rules.

Hence, we have to deal with ternary environments. However, we wish to derive a boolean transition relation for the micro step behavior of a Quartz program that can be analyzed by state of the art verification techniques. For this reason, we embed the unknown value $\bot$ explicity as follows: For every
local, reincarnated local, and output variable $x$, we add a boolean-typed variable $x^\perp$ that holds iff the value of $x$ is known. Hence, the pair $(x^\perp, x)$ encodes the actual value of the variable $x$: if $x^\perp$ holds, then the value of $x$ is known and is contained in $x$, otherwise the value of $x$ is not yet known and we have to ignore the content of $x$.

Using the variables $x^\perp$, we can explicitly model the progress of the information flow that is obtained by evaluating the program expression step by step until either assignments can be executed that determine the current value of a variable or until it becomes clear that no assignment will modify the current value of a variable so that the reaction to absence will determine it.

This progress of the information flow also makes use of lazy evaluation as explained in Figure 4.1 that has to be encoded in the transition relation at the micro step level. To this end, we formally define a function that maps a program expression $\sigma$ of arbitrary type to a boolean formula $k(\sigma)$ such that $k(\sigma)$ holds iff the expression $\sigma$ can be evaluated to a known value. Formally, the formula $k(\sigma)$ follows the rules of Figure 4.1 and is defined as shown below:

**Definition 4.17 (Known Status of Expressions).** For every expression $\sigma$, we define a formula $k(\sigma)$ recursively as follows:

- **for variables and constants, we define**
  - $k(x) := \{\begin{array}{ll} true & : \text{if } x \text{ is an input or location variable} \\ x^\perp & : \text{if } x \text{ is a local, reincarnated, or output variable} \end{array}$
  - $k(c) := c$

- **for tuple constructor and selector, we define**
  - $k((\tau_{n-1}, \ldots, \tau_0)) := k(\tau_{n-1}) \land \ldots \land k(\tau_0)$
  - $k(\tau.\pi) := k(\tau) \land k(\pi)$

- **for array access, we define**
  - $k(x[\tau]) := k(x) \land k(\pi)$

- **for bitwise operators, we define:**
  - $k(\varphi \oplus \psi) := k(\varphi) \land k(\psi)$
  - $k(\varphi\{\pi_1\}) := k(\varphi) \land k(\pi_1)$
  - $k(\varphi\{\pi_1, \pi_2\}) := k(\varphi) \land k(\pi_1) \land k(\pi_2)$
  - $k(\varphi\{\pi_1\}) := k(\varphi) \land k(\pi_1)$
  - $k(\varphi\{\pi_2\}) := k(\varphi) \land k(\pi_2)$
  - $k(\text{fromArray}(x)) := k(x)$
  - $k(\text{reverse}(\varphi)) := k(\varphi)$
  - $k(\{\tau :: n\}) := k(\tau)$
  - $k(\text{sizeOf}(\varphi)) := \text{true}$
\[ k(\varphi \rightarrow \psi) := k(!\varphi \parallel \psi) \]
\[ k(\varphi \leftrightarrow \psi) := k(!\varphi \parallel \psi) \land k(\psi \parallel \varphi) \]
\[ k(\varphi \text{ xor } \psi) := k(!((\varphi \leftrightarrow \psi))) \]

- for explicit type converters, we define:
  \[ k(\text{int2bv}(\tau)) := k(\tau) \]
  \[ k(\text{nat2bv}(\tau)) := k(\tau) \]
  \[ k(\text{bv2int}(\tau)) := k(\tau) \]
  \[ k(\text{bv2nat}(\tau)) := k(\tau) \]

- for arithmetic operators, we define:
  \[ k(-\tau) := k(\tau) \]
  \[ k(\text{abs}(\tau)) := k(\tau) \]
  \[ k(\tau + \pi) := k(\tau) \land k(\pi) \]
  \[ k(\tau - \pi) := \begin{cases} k(\tau) \land k(\pi) & : \text{if } \tau \text{ and } \pi \text{ are of some type nat<n>} \\ k(\tau) \land k(\pi) & : \text{otherwise} \end{cases} \]
  \[ k(\tau \ast \pi) := k(\tau) \land k(\pi) \land \tau = 0 \lor k(\pi) \land \pi = 0 \]
  \[ k(\tau / \pi) := k(\tau) \land k(\pi) \land k(\pi) \land \pi = 0 \]
  \[ k(\exp2(\tau)) := k(\tau) \]
  \[ k(\log2(\tau)) := k(\tau) \]
  \[ k(\text{sat<n>}(\tau)) := k(\tau) \]

- for arithmetic relations, we define:
  \[ k(\tau \leq \pi) := \begin{cases} k(\tau) \land (\tau = 0 \lor k(\pi) \land \tau \leq \pi) & : \text{if } \tau \text{ and } \pi \text{ are of some type nat<n>} \\ k(\pi) \land k(\tau) & : \text{otherwise} \end{cases} \]
  \[ k(\tau < \pi) := k(\tau \leq \pi) \land k(\pi) \land k(\pi) \land \tau = 0 \lor k(\tau) \land k(\pi) \land \tau \leq \pi \]
  \[ k(\tau > \pi) := k(\tau \leq \pi) \land k(\pi) \land k(\tau) \land k(\pi) \land \tau = 0 \lor k(\pi) \land k(\tau) \land \tau \leq \pi \]
  \[ k(\tau \geq \pi) := k(\tau \leq \pi) \land k(\tau) \land k(\pi) \land k(\pi) \land k(\tau) \land k(\pi) \land \tau = 0 \lor k(\pi) \land k(\tau) \land k(\pi) \land \tau \leq \pi \]

- for equality and inequality operators, we define:
  \[ k(\tau = \pi) := k(\tau) \land k(\pi) \land k(\tau) \land k(\pi) \land \tau = 0 \lor k(\pi) \land k(\tau) \land k(\pi) \land \tau \leq \pi \]

- finally,
  \[ k((\varphi ? \tau : \pi)) := \begin{cases} k(\varphi) \land \varphi \land \tau \lor k(\varphi) \land \neg \varphi \land \pi \lor k(\tau) \land k(\pi) \land \tau = \pi & : \text{if } \varphi \text{ and } \tau \text{ are of some type nat<n>} \\ k(\tau) \land k(\pi) \land \tau = \pi & : \text{otherwise} \end{cases} \]
  \[ k(\text{sizeOf}(\tau)) := k(\tau) \]

Clearly, in the behavior of the program, we can only make use of an expression if we know its value. For this reason, the entire execution of the actions is controlled by the data flow. In Chapter 6, we therefore consider the interpretation of Quartz programs in terms of data flow machines that completely correspond to the data flow computers that have been built in the eighties.

As the reaction of a program is determined in micro steps that are executed in several steps of a macro step, we have to explicitly distinguish between micro and macro steps of the execution. This distinction has also been made in the SOS transition rules, while the SOS reaction rules only refer to the current macro step. For this reason, we introduce a clock signal clock, that is
true whenever all variables have become known values\textsuperscript{2}. If this happens, the delayed assignments are executed, the location variables change their values according to the control flow, and the input variables are allowed to change their values in a nondeterministic way.

Macro steps are therefore defined in terms of occurrences of the clock signal, and between two clock ticks, the micro steps of a macro step are executed: As long as clock is false, the immediate assignments to the variables are executed if the values of the guards and right hand side expressions are known, and the guard is true.

We therefore distinguish between the \textit{information flow} and the \textit{data flow}. The information flow of a variable $x$ is determined by the corresponding variable $k(x) = x^\perp$ that holds if and only if the value of $x$ is already determined in the current macro step.

The transition relation of the information flow can be formulated as an equation system as shown in Figure 4.11 that contains the following cases:

- If there is no clock tick, the value of $x$ remains known if it was already known\textsuperscript{3}.
- If there is no clock tick, the value of $x$ becomes known if a guarded action $(\gamma_j, x=\tau_j)$ with an immediate assignment $x=\tau_j$ can be fired. This is the case if and only if the value of the guard $\gamma_j$ is known to be true and if the value of the right hand side expression $\tau_j$ is known.
- If there is no clock tick, and all guards $\gamma_j$ are known to be false, the reaction to absence determines the value of $x$: The formula of Figure 4.11 simply demands that $\text{next}(x)=x$ has to hold in this case, since $x$ has been given the now desired value at the previous clock tick as a preliminary value.
- If there is a clock tick, then the values of all variables are known. Therefore, we can execute all enabled delayed actions. If one of the delayed actions can be fired, then the value of $x$ is known for the following macro step.
- Otherwise, the value of $x$ is not known.

The data flow of $x$ is determined by the same cases as formalized in formula $\text{ValTrans}_x$ given in Figure 4.11:

- If there is no clock tick and a guarded action $(\gamma_j, x=\tau_j)$ with an immediate assignment $x=\tau_j$ can be fired, then $x$ will receive the value of $\tau_j$ at the next point of time. Note that if more than one guarded action can be fired with different values $\tau_i$ and $\tau_j$, then there is no transition.

\textsuperscript{2} Actually, we have to delay the clock signal by a further step, as will be described later.

\textsuperscript{3} Although a guarded action $(\gamma_j, x=\tau_j)$ that once fires in a macro step, remains enabled until the end of the macro step, we have to add this constraint, since the variable $x$ may be known due to a delayed assignment of the previous macro step.
Computed Guarded Commands of $x$ and its Reincarnations $x_1, \ldots, x_d$:

$(\chi_1, \text{next}(x) = \pi_1), \ldots, (\chi_q, \text{next}(x) = \pi_q)$

$(\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p)$

$(\gamma_1, x_1 = \tau_{1,1}), \ldots, (\gamma_{1,p_1}, x_1 = \tau_{1,p_1})$

$\vdots$

$(\gamma_{d,1}, x_d = \tau_{d,1}), \ldots, (\gamma_{d,p_d}, x_d = \tau_{d,p_d})$

Propagation of Knowledge During Clock Cycle:

$\text{KnownTrans}_{x_i} := \left( \text{next}(k(x_i)) :\iff \neg \text{clock} \land \left( k(x_i) \lor \left( \bigwedge_{j=1}^{p_i} \left( k(\gamma_{i,j}) \land \gamma_{i,j} \land k(\tau_{i,j}) \right) \lor \left( \bigwedge_{j=1}^{p_i} k(\gamma_{i,j}) \land \neg \gamma_{i,j} \right) \right) \right) \right)$

$\text{KnownTrans}_{x} := \left( \text{next}(k(x)) :\iff \neg \text{clock} \land \left( k(x) \lor \left( \bigwedge_{j=1}^{p_i} \left( k(\gamma_{j}) \land \gamma_{j} \land k(\tau_{j}) \right) \lor \left( \bigwedge_{j=1}^{p_i} k(\gamma_{j}) \land \neg \gamma_{j} \right) \right) \lor \left( \text{clock} \land \bigvee_{j=1}^{p_i} \pi_{j} \right) \right) \right)$

Micro Step Transition Relation of Reincarnated Variable $x_j$:

$\text{ReincarTrans}_{x_j} := \left( \begin{array}{c} \neg \text{clock} \to \left( \bigwedge_{j=1}^{p_i} \left( k(\gamma_{i,j}) \land \gamma_{i,j} \land k(\tau_{i,j}) \right) \rightarrow \text{next}(x_j) = \tau_{i,j} \right) \right) \land \\
\left( \neg \text{clock} \to \left( \neg \left( \bigvee_{j=1}^{p_i} k(\gamma_{j}) \land \gamma_{j} \land k(\tau_{j}) \right) \rightarrow \text{next}(x_j) = \text{Default}(x) \right) \right) \land \\
\left( \text{clock} \rightarrow \text{next}(x_j) = \text{Default}(x) \right) \end{array} \right)$

Micro Step Transition Relation of Local/Output Variable $x$:

$\text{ValTrans}_x := \left( \begin{array}{c} \neg \text{clock} \to \left( \bigwedge_{j=1}^{q} \left( x_j \rightarrow \text{next}(x) = \pi_j \right) \right) \land \\
\neg \text{clock} \to \left( \neg \left( \bigvee_{j=1}^{q} k(\gamma_{j}) \land \gamma_{j} \land k(\tau_{j}) \right) \rightarrow \text{next}(x) = x \right) \right) \land \\
\left( \text{clock} \rightarrow \left( \bigwedge_{j=1}^{q} \left( \gamma_{j} \rightarrow \text{next}(x) = \pi_j \right) \right) \right) \land \\
\left( \text{clock} \rightarrow \left( \neg \left( \bigvee_{j=1}^{q} x_j \rightarrow \text{next}(x) = \text{Initialize}(x) \right) \right) \right) \end{array} \right)$

where $\text{Initialize}(x)$ is defined as in Figure 4.10 on Page 126

Fig. 4.11. Micro Step Transition Relation to Define the Data Flow of a Local/Output Variable $x$
• If there is no clock tick and no immediate guarded action can be fired, then \( x \) will keep its value. This covers several cases: First, if the reaction-to-absence should take place, since all guards \( \gamma_j \) are known to be false, then keeping the value of \( x \) is correct, since we already provided the desired value for \( x \) at the previous clock tick: Either there was a delayed assignment in the previous macro step that determined the current value of \( x \), or \( x \) has been initialized according to \( \text{Initialize}(x) \).

• If there is a clock tick, then the values of all variables are known, and therefore, we can execute all enabled delayed actions. If one of the delayed actions can be fired, then the value of \( x \) is known for the following macro step. Note again, that there is no transition if several delayed guarded actions with different values \( \pi_i \) and \( \pi_j \) are fired.

• Finally, if there is a clock tick, but no delayed action can be fired, then \( x \) is initialized according to \( \text{Initialize}(x) \).

Note that reincarnated variables do always have event storage mode and no delayed actions, which simplifies their transition relation accordingly. Finally, note that the case distinctions as given by the left hand sides of the implications of \( \text{ValTrans}_x \) and \( \text{ReincarTrans}_x \) are complete; that is the disjunction of the left hand sides of the implications is valid.

The formulas given in Figure 4.11 describe the micro step transition relation of a particular local/output variable \( x \) together with its potential reincarnated variables. In addition to this, we also have to determine the flow of the input and location variables, and we have to define the clock signal. The latter is defined as follows:

\[
\text{ClockTrans} \equiv \left( \text{next(clock)} :\Leftrightarrow \bigwedge_{x} k(x) \land \bigwedge_{j=1}^{d} k(x_j) \right)
\]

In principle, the new clock tick can arrive as soon as all values of the local and output variables become known. This suggest to define the clock signal simply as an abbreviation of the right hand side of the above transition equation. However, this would hide certain write conflicts: Assume that the value of a variable \( x \) becomes known in the last micro step of the macro step. Hence, after this point of time (in terms of micro steps), all variables have known values. Therefore, a so-far not known guard may now also become known and the execution of its assignment may lead to a write conflict to the already determined value of \( x \). Hence, we have to add a further micro step before the clock tick so that all enabled guarded actions will fire at that point of time.

When the clock tick arrives, we perform the following actions to initiate the next macro step:

• The values of the location variables \( \ell \) can be computed in terms of the now known value of the inputs, outputs, and local variables. Hence, we add for every location variable \( \ell \) with transition equation \( \text{next}(\ell) = \varphi_{\ell} \) the following transition equation \( \text{next}(\ell) = (\text{clock} \Rightarrow \varphi_{\ell} | \ell) \).
• The input variables are read from the environment, which is done by allowing the input variables to change their values in a nondeterministic way. However, when clock is false, we have to make sure that the inputs do not change any more. Therefore, we have to add the formula \( \neg \text{clock} \rightarrow \text{next}(x) = x \) for every input variable \( x \).

It is easily seen that the micro step behavior as formalized in Figure 4.11 and in the above additional explanations for the clock definition, the location variables and the input variables describe the semantics of an asynchronous circuit. We consider the relationship to asynchronous hardware circuits in more detail in Section ??.

Moreover, the entire behavior is described in terms of conditional rewrite rules, so that also term rewriting techniques are adequate for the analysis of the micro step behavior of Quartz programs.
It is interesting to consider the classic case, i.e., the case where \( x \) is an event variable that only has immediate assignments with the right hand side \( \text{true} \). In this case, the formulas of Figure 4.11 reduce to the ones shown in Figure 4.12.

**Computed Guarded Commands of \( x \) and its Reincarnations \( x_1, \ldots, x_d \):**

\[
(\gamma_1, x=\text{true}), \ldots, (\gamma_p, x=\text{true}),
(\gamma_{1,1}, x_1=\text{true}), \ldots, (\gamma_{1,p_1}, x_1=\text{true}),
\vdots,
(\gamma_{d,1}, x_d=\text{true}), \ldots, (\gamma_{d,p_d}, x_d=\text{true})
\]

**Propagation of Knowledge During Clock Cycle:**

\[
\text{KnownTrans}_{x_i} \equiv \left( \text{next}(k(x_i)) :\leftrightarrow \neg\text{clock} \land \left( \bigvee_{j=1}^{p_i} k(\gamma_{i,j}) \land \gamma_{i,j} \lor \bigwedge_{j=1}^{p_i} k(\gamma_{i,j}) \land \neg\gamma_{i,j} \right) \right)
\]

\[
\text{KnownTrans}_x \equiv \left( \text{next}(k(x)) :\leftrightarrow \neg\text{clock} \land \left( \bigvee_{j=1}^{p} k(\gamma_{j}) \land \gamma_{j} \lor \bigwedge_{j=1}^{p} k(\gamma_{j}) \land \neg\gamma_{j} \right) \right)
\]

**Micro Step Transition Relation of Reincarnated Variable \( x_j \):**

\[
\text{ReincarTrans}_{x_j} \equiv \left( \text{next}(x_j) :\leftrightarrow \neg\text{clock} \land \left( \bigwedge_{j=1}^{p_j} k(\gamma_{j,j}) \land \gamma_{j,j} \right) \right)
\]

**Micro Step Transition Relation of Local/Output Variable \( x \):**

\[
\text{ValTrans}_x \equiv \left( \text{next}(x) :\leftrightarrow \neg\text{clock} \land \left( \bigwedge_{j=1}^{p} k(\gamma_{j}) \land \gamma_{j} \right) \right)
\]

**Fig. 4.12.** Micro Step Transition Relation for Event Variable \( x \) with only Immediate Assignments \( x=\text{true} \)

As can be seen in Figure 4.12, the transition relation can now be given in form of an equation system. The reason for this is that at every clock tick, the known values as well as the values themselves are reset to false, and during the clock cycle, the value of such an event signal becomes true if and only if one of the guards \( \gamma_j \) is known to be true. The known value becomes additionally true if all guards \( \gamma_j \) are known to be false. As a consequence \( x \) implies \( k(x) \).

The equational form given in Figure 4.12 is interesting because it gives a new insight in the causality analysis in terms of a fixpoint computation:

The reason why we can obtain an equational form is that there are by definition no write conflicts if we only have assignments of the form \( x=\text{true} \). In principle, we can therefore also allow delayed assignments of the form \( \text{next}(x)=\text{true} \), and we could still derive equation systems as it has been shown in [226]. However, this requires the introduction of new state-holding
variables since we have to store if a delayed assignment was enabled in the previous macro step.

In the general case, we have to consider the possibility that several assignments are enabled at the same point of time and that these assignments refer to different values.
4.6 Symbolic SOS Reaction Rules

4.6.2 Can- and Must-Guarded Actions

Recall that the SOS reaction rules have been primarily used to compute the sets \( D_{\text{must}} \) and \( D_{\text{can}} \) of immediate assignments that must be and can be executed in the current reaction step. To this end, we had also to estimate the instantaneity of the statements by the boolean values \( t_{\text{must}} \) and \( t_{\text{can}} \) as defined in Section 4.3.

A symbolic version of the SOS reaction rules can be obtained by computing the guarded actions that must be and can be executed in the current reaction. The surprising fact is that there is nothing we have to compute for this purpose. Instead, all we have to do is to interpret the guards of \( \text{ActSurf}_h(\text{true}, S) \) in a three-valued setting as explained in Section 4.1. However, this is not sufficient, since we also have to incorporate the reaction-to-absence.

For this reason, the symbolic causality analysis is in principle a symbolic execution of the interpreter function given in Section 4.4. However, we are not yet ready to describe such a symbolic execution, since the symbolic versions of the control and data flow that we have derived in this section are not executable. ‘Executable’ means here that we only have formulas that can check the satisfiability of given assignments, but we can not determine the assignment in an incremental fashion as done by the interpreter function given in Section 4.4. We will see in the next chapter that we can refine the symbolic control and data flow presentation developed in this Chapter so that a symbolic execution as required by the symbolic causality analysis will be possible. Hence, we have to postpone the symbolic version of the SOS reaction rules to the following Chapter, where it is then shown that it is equivalent to the ternary simulation of hardware circuits.

To formally reason about a symbolic causality analysis, we have to deal with incomplete environments \( \mathcal{E} \) as defined in Section 4.1. In particular, we consider the following sets of actions:

**Definition 4.18 (Symbolic Causality Analysis).** For a given Quartz statement \( S \), and an incomplete environment \( (\mathcal{E}, h) \), we define the following sets of actions:
Proof. The evaluation of expression is monotonic. Therefore, we have for all $\alpha \in \text{MustAct}_{(E, h)}(S) := \{\alpha \mid \exists \gamma. (\gamma, \alpha) \in \text{ActSurf}_{h}(true, S) \wedge \llbracket \gamma \rrbracket_{E}^{h} \neq false\}$

Since the environment $(E, h)$ is incomplete, the evaluation $\llbracket \gamma \rrbracket_{E}^{h}$ may be either one of $\{\perp, true, false\}$. Therefore, the set $\text{CanAct}_{(E, h)}(S)$ contains those actions with unclear precondition $\gamma$, so that there is still some hope that $\gamma$ could become finally true. In contrast, $\text{MustAct}_{(E, h)}(S)$ contains only those actions where the precondition $\gamma$ can not become false, so there is no risk that $\gamma$ could become finally false.

Since the environments are monotonically updated with respect to the partial order of the environments $E$ as defined in Definition 4.3 on page 88, we can alternatively use the following ‘characterizations’ of the above sets:

**Lemma 4.19 (Symbolic Causality Analysis).** For every Quartz statement $S$, and every incomplete environment $(E, h)$, we have the following inclusions:

- $\text{CanAct}_{(E, h)}(S) \supseteq \{\alpha \mid \exists \gamma. \exists E'. (\gamma, \alpha) \in \text{ActSurf}_{h}(true, S) \land (E, h) \sqsubseteq (E', h) \land \llbracket \gamma \rrbracket_{E'}^{h} = true\}$
- $\text{MustAct}_{(E, h)}(S) \subseteq \{\alpha \mid \exists \gamma. \forall E'. (\gamma, \alpha) \in \text{ActSurf}_{h}(true, S) \land (E, h) \sqsubseteq (E', h) \land \llbracket \gamma \rrbracket_{E'}^{h} = true\}$

Proof. The evaluation of expression is monotonic. Therefore, we have for all $(E, h) \sqsubseteq (E', h)$ the following implications:

- $\llbracket \gamma \rrbracket_{E}^{h} = true$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} = true$
- $\llbracket \gamma \rrbracket_{E}^{h} = false$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} = false$
- $\llbracket \gamma \rrbracket_{E}^{h} = \perp$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} \in \{\perp, true, false\}$
- $\llbracket \gamma \rrbracket_{E'}^{h'} = true$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} = false$
- $\llbracket \gamma \rrbracket_{E'}^{h'} = false$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} = true$
- $\llbracket \gamma \rrbracket_{E}^{h} = \perp$ implies $\llbracket \gamma \rrbracket_{E'}^{h'} = \perp$

Consider now the must-actions: If already $\llbracket \gamma \rrbracket_{E}^{h} = true$ holds, then we must also have $\llbracket \gamma \rrbracket_{E'}^{h'} = true$ for every environment $(E', h')$ with $(E, h) \sqsubseteq (E', h)$. The converse may however not be true, since the evaluation of $\llbracket \gamma \rrbracket_{E}^{h}$ may not consider ‘global’ effects in the evaluation of $\gamma$: Consider, e.g., $\gamma = (x+y) - x = y$, which is true for all values of $x$ and $y$. However, when one of these value is not yet known, then the result of the evaluation is $\perp$.

Analogously, if $\llbracket \gamma \rrbracket_{E'}^{h'} = true$ holds, then we also have $\llbracket \gamma \rrbracket_{E}^{h} \neq false$ for all $(E, h)$ with $(E, h) \sqsubseteq (E', h)$. Hence, the mentioned inclusion holds, and the converse is not the case for the same reason as mentioned in the case of the must-actions. \qed

Hence, our definitions are actually a bit too strong and can be liberated as given by the ‘alternative characterizations’ of the above lemma (that are however more difficult to check). Nevertheless, we can derive some interesting facts that will lead to further discussions. In particular, we can derive the
recursive laws for the computation of $\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$ and $\text{MustAct}_{(\mathcal{E},\mathcal{H})}(S)$ as given in the Figures 4.13 and 4.14.

There is an asymmetry for the definition of the actions of the if-statements: While the can-actions take the union of the can-actions of the two substatements, the must-actions (which are supposed to be dual) do not take the intersections. The reason for this behavior is as follows:

---

Fig. 4.13. Possibly Executed Actions

- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}(x=\tau) = \{x=\tau\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((x)=\tau) = \{\text{next}(x)=\tau\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{assume})(\sigma)) = \{\text{assume}(\sigma)\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{assert})(\sigma)) = \{\text{assert}(\sigma)\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{nothing})) = \{\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\ell:\text{pause})) = \{\}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{if}(\sigma) S_1 \text{ else } S_2))$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_1) : \text{if } [\sigma]^h = \text{true} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_2) : \text{if } [\sigma]^h = \text{false} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_1) \cup \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_2) : \text{if } [\sigma]^h = \bot \}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\{S_1; S_2\}))$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_1) : \text{if } \lbrack \text{inst}_h(S_1) \rbrack^h = \text{false} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_1) \cup \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_2) : \text{otherwise} \}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\{S_1 \parallel S_2\})) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_1) \cup \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S_2)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{do } S \text{ while}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{while}(\sigma) S))$
  - $\{\{\} : \text{if } [\sigma]^h = \text{false} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S) : \text{otherwise} \}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{abort } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{weak} \text{ abort } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{weak immediate abort } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{immediate abort } S \text{ when}(\sigma)))$
  - $\{\{\} : \text{if } [\sigma]^h = \text{true} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S) : \text{if } [\sigma]^h \in \{\bot, \text{false} \} \}$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{suspend } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{weak} \text{ suspend } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{weak immediate suspend } S \text{ when}(\sigma))) = \text{CanAct}_{(\mathcal{E},\mathcal{H})}(S)$
- $\text{CanAct}_{(\mathcal{E},\mathcal{H})}((\text{immediate suspend } S \text{ when}(\sigma)))$
  - $\{\{\} : \text{if } [\sigma]^h = \text{true} \}$
  - $\{\text{CanAct}_{(\mathcal{E},\mathcal{H})}(S) : \text{if } [\sigma]^h \in \{\bot, \text{false} \} \}$
alternatives have been discussed in [229]:

For example, the following definition of causality depends on some subtle definitions that are from a logical point of view equivalent, and redundant. For example, the following alternatives have been discussed in [229]:

\[
\begin{align*}
\text{MustAct}_{(E,h)}(x=\tau) & = \{ x=\tau \} \\
\text{MustAct}_{(E,h)}(\text{next}(x)=\tau) & = \{ \text{next}(x)=\tau \} \\
\text{MustAct}_{(E,h)}(\text{assume}(\sigma)) & = \{ \text{assume}(\sigma) \} \\
\text{MustAct}_{(E,h)}(\text{assert}(\sigma)) & = \{ \text{assert}(\sigma) \} \\
\text{MustAct}_{(E,h)}(\text{nothing}) & = \{ \} \\
\text{MustAct}_{(E,h)}(\text{t:pause}) & = \{ \}
\end{align*}
\]

\[
\begin{align*}
\text{MustAct}_{(E,h)}(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) & = \\
& = \left\{ \begin{array}{l}
\text{MustAct}_{(E,h)}(S_1) : \text{if } [\sigma]^h_E = \text{true} \\
\text{MustAct}_{(E,h)}(S_2) : \text{if } [\sigma]^h_E = \text{false} \\
\{ \} : \text{if } [\sigma]^h_E = \bot
\end{array} \right\}
\end{align*}
\]

\[
\begin{align*}
\text{MustAct}_{(E,h)}(\{S_1;S_2\}) & = \\
& = \{ \begin{array}{l}
\text{MustAct}_{(E,h)}(S_1) \cup \text{MustAct}_{(E,h)}(S_2) : \text{if } \mathcal{E}(\text{inst}_h(S_1)) = \text{true} \\
\text{MustAct}_{(E,h)}(S_1) : \text{if } \mathcal{E}(\text{inst}_h(S_1)) \in \{ \bot, \text{true} \}
\end{array} \}
\end{align*}
\]

\[
\begin{align*}
\text{MustAct}_{(E,h)}(\text{do } S \text{ while}(\sigma)) & = \text{MustAct}_{(E,h)}(S) \\
\text{MustAct}_{(E,h)}(\text{for} \ S \text{ when}(\sigma)) & = \text{MustAct}_{(E,h)}(S) \\
\text{MustAct}_{(E,h)}(\text{while}(\sigma) \ S) & = \{ \begin{array}{l}
\text{MustAct}_{(E,h)}(S) : \text{if } [\sigma]^h_E = \text{true} \\
\{ \} : \text{otherwise}
\end{array} \}
\end{align*}
\]

\[
\begin{align*}
\text{MustAct}_{(E,h)}(\text{abort } S \text{ when}(\sigma)) & = \text{MustAct}_{(E,h)}(S) \\
\text{MustAct}_{(E,h)}(\text{weak} \ S \text{ when}(\sigma)) & = \text{MustAct}_{(E,h)}(S) \\
\text{MustAct}_{(E,h)}(\text{weak immediate abort } S \text{ when}(\sigma)) & = \text{MustAct}_{(E,h)}(S) \\
\text{MustAct}_{(E,h)}(\text{immediate abort } S \text{ when}(\sigma)) & = \{ \begin{array}{l}
\text{MustAct}_{(E,h)}(S) : \text{if } [\sigma]^h_E = \text{false} \\
\{ \} : \text{if } \mathcal{E}(\text{inst}_h(S_1)) \in \{ \bot, \text{true} \}
\end{array} \}
\end{align*}
\]

\[
\begin{align*}
\text{Necessarily Executed Actions}
\end{align*}
\]

The definition of causality depends on some subtle definitions that are from a logical point of view equivalent, and redundant. For example, the following alternative has been discussed in [229]:

\[
\begin{align*}
\text{MustAct}_{(E,h)}(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) & = \{ \alpha \mid \exists \gamma.(\gamma, \alpha) \in \text{ActSurf}_h(\text{true}, \text{if}(\sigma) \ S_1 \ \text{else} \ S_2) \land [\gamma]^h_E = \text{true} \} \\
& = \{ \alpha \mid \exists \gamma.(\gamma, \alpha) \in \text{ActSurf}_h(\sigma, S_1) \cup \text{ActSurf}_h(\neg\sigma, S_2) \land [\gamma]^h_E = \text{true} \} \\
& = \{ \alpha \mid \exists \gamma.(\gamma, \alpha) \in \text{ActSurf}_h(\text{true}, S_1) \land [\gamma]^h_E = \text{true} \} \\
& = \{ \alpha \mid \exists \gamma.(\gamma, \alpha) \in \text{ActSurf}_h(\text{true}, S_1) \land \mathcal{E}(\gamma \land \sigma) = \text{true} \} \\
& = \{ \alpha \mid \exists \gamma.(\gamma, \alpha) \in \text{ActSurf}_h(\text{true}, S_1) \land \mathcal{E}(\gamma \land \neg\sigma) = \text{true} \} \\
& = \{ \text{MustAct}_{(E,h)}(S_1) : \text{if } [\sigma]^h_E = \text{true} \\
& \{ \} : \text{if } [\sigma]^h_E = \bot
\end{align*}
\]
The instantaneity of if-statements can be defined as follows:
- \( \text{inst}_h(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) = h(\sigma) \land \text{inst}_h(S_1) \lor \neg h(\sigma) \land \text{inst}_h(S_2) \)
- \( \text{inst}_h(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) = h(\sigma) \land \text{inst}_h(S_1) \lor \neg h(\sigma) \land \text{inst}_h(S_2) \lor \text{inst}_h(S_1) \land \text{inst}_h(S_2) \)

From a logical point of view, both expressions are equivalent, since the boolean formulas \( a \land b \lor \neg a \land c \) and \( a \land b \lor \neg a \land c \lor b \land c \) are equivalent. The latter formula contains just the redundant prime implicant \( b \land c \). Using either the one or the other definition, however, changes the set of construction programs. For example, program P10 below becomes constructive:

```c
module P10(event &o) {
    if(o) nothing;
    emit o;
}
```

Using the first variant, the guard of `emit o` is either \( o \land true \lor \neg o \land true \), hence, \( o \lor \neg o \), while the second variant yields \( o \land true \lor \neg o \land true \lor true \land true \), hence, `true`. Therefore, adding the redundancy makes more programs constructive.

In a similar way, we can also refine the definition of the surface actions. Here are the two alternatives:
- \( \text{ActSurf}_h(\varphi, \text{if}(\sigma) \ S_1 \ \text{else} \ S_2) := \text{ActSurf}_h(\varphi \land h(\sigma), S_1) \cup \text{ActSurf}_h(\varphi \land \neg h(\sigma), S_2) \)
- \( \text{ActSurf}_h(\varphi, \text{if}(\sigma) \ S_1 \ \text{else} \ S_2) := A_1 \cup A_2 \cup \{(\gamma_1 \land \gamma_2, \mathcal{C}) \mid (\gamma_1, \mathcal{C}) \in A_1 \land (\gamma_2, \mathcal{C}) \in A_2\} \), where \( A_1 := \text{ActSurf}_h(\varphi \land h(\sigma), S_1) \) and \( A_2 := \text{ActSurf}_h(\varphi \land \neg h(\sigma), S_2) \)

The idea of the latter definition is that actions \( \mathcal{C} \) must be executed when this action has two guards \( \gamma_1 \) and \( \gamma_2 \) that stem from the different branches, and both guards must be true. Again, this makes some programs constructive that would otherwise be not constructive. For example, the simple program P12 becomes constructive:

```c
module P12(event &o) {
    if(o) emit o;
    else emit o;
}
```

In [53] and [229], some more alternatives have been discussed. From a logical point of view, it is even known that there is a maximal notion of causality that is related to Eichelberger’s prime implicants theorem [94] to avoid hazards in asynchronous circuits.

We can also discuss the rules of Figures 4.6 and 4.6 in the light of the three-valued logic. To this end, it is convenient to encode the three logic values \( \perp \), `true`, and `false` by a pair of boolean values. There are many encodings that
could be used, however, the following negated dual-rail\(^4\) encoding has some convenient properties:

<table>
<thead>
<tr>
<th>(x \in \mathbb{B}^2)</th>
<th>(\ell(x))</th>
<th>(\land)</th>
<th>(\lor)</th>
<th>(\neg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(false, true)</td>
<td>⊥</td>
<td>(⊥)</td>
<td>(⊥)</td>
<td>(⊥)</td>
</tr>
<tr>
<td>(false, false)</td>
<td>0</td>
<td>0 0 0  ⊥</td>
<td>(⊥)</td>
<td>(0)</td>
</tr>
<tr>
<td>(true, true)</td>
<td>1</td>
<td>(1) (1) (1) (1)</td>
<td>(1) (0) (1) (0)</td>
<td>1 0</td>
</tr>
<tr>
<td>(true, false)</td>
<td>⊤</td>
<td>(⊥)</td>
<td>(⊥)</td>
<td>(⊥)</td>
</tr>
</tbody>
</table>

For the above dual-rail encoding of true, false, and \(\perp\) by two boolean values \((b_1, b_2)\), and the three-valued boolean operations \(\land\), \(\lor\), and \(\neg\), we can define the three-valued boolean operations \(\land\), \(\lor\), and \(\neg\) directly in terms of the boolean pairs that encode the three-valued truth values. It can be easily verified that the following holds:

- \((a_1, a_2) \land (b_1, b_2) = (a_1 \land b_1, a_2 \land b_2)\)
- \((a_1, a_2) \lor (b_1, b_2) = (a_1 \lor b_1, a_2 \lor b_2)\)
- \(\neg(a_1, a_2) = (\neg a_2, \neg a_1)\)

Now, consider the pair \((t_{\text{must}}, t_{\text{can}})\) of the boolean flags that have been used in the definition of the SOS reaction rules. It is interesting to re-read the SOS reaction rules in terms of the three-valued interpretation of this pair.

For example, consider the conditional statement \(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2\) and assume that \(i_1\) and \(i_2\) are the three-valued conditions for instantaneous execution of \(S_1\) and \(S_2\), respectively. According to the rules given in Figure 4.6, we can easily check that they define the instantaneous execution of the conditional statement as \(\sigma \land i_1 \lor \neg \sigma \land i_2 \lor i_1 \land i_2\). To see this, recall that we encode \(i_1\) and \(i_2\) by the pairs \((t_{\text{must}}, t_{\text{can}})\) and \((t_{\text{must}}^2, t_{\text{can}}^2)\), respectively. For \(\sigma = (\sigma_1, \sigma_2)\), we therefore obtain the following two rails:

- \(\sigma_1 \land t_{\text{must}}^1 \lor \neg \sigma_2 \land t_{\text{must}}^2 \lor t_{\text{must}}^1 \land t_{\text{must}}^2\)
- \(\sigma_2 \land t_{\text{can}}^1 \lor \neg \sigma_1 \land t_{\text{can}}^2 \lor t_{\text{can}}^1 \land t_{\text{can}}^2\)

It is now easy to see that for the three cases \((\sigma_1, \sigma_2) \in \{(\text{false, false}), (\text{false, true}), (\text{true, true})\}\), we obtain exactly the definitions as stated in Figure 4.6.

In case of sequences \(S_1; S_2\), the condition for instantaneous execution is \(i_1 \land i_2\) where \(i_1\) and \(i_2\) are the three-valued conditions for instantaneous execution of \(S_1\) and \(S_2\), respectively. If these are again encoded by \((t_{\text{must}}^1, t_{\text{can}}^1)\) and \((t_{\text{must}}^2, t_{\text{can}}^2)\), respectively, we obtain the definitions as given in Figure 4.6.

\(^4\) The encoding we use here is related to the well-known dual-rail encoding. The difference to dual-rail encoding is that the second pair is negated.
In the previous section, we have defined an operational semantics in terms of SOS transition and reaction rules. This is not only necessary to define the semantics of the language in a precise way; it is also useful to implement a simulator or an interpreter for the language. However, it is inconvenient to handle all possible inputs at compile time which is necessary for a compiler. For this reason, we have already considered the symbolic form of the SOS rules that is given in terms of the symbolic control flow predicates and the guarded actions. While this symbolic form solves the problem to consider all possible inputs at once, it does not lead to an algorithm to compile the programs into executable code.

This synthesis procedure is based on various prerequisites that have developed in previous work. The starting point was the translation to hardware circuits given in [222] that has been formally verified using the HOL theorem prover. To this end, the control flow predicates have been defined [223, 224, 233] as a symbolic form of the operational semantics as described in the previous chapter. As schizophrenia problems have not been considered in these previous attempts, a generalization to handle all kinds of schizophrenia has been developed in [227, 228]. To obtain a translation that runs in quadratic time, subterms that are generated during the translation have to be abbreviated by new variables. In the former version described in [227], we used two equation systems $E_{srfc}$ and $E_{dpth}$ that contained the abbreviations made for the surface and the depth, respectively.

This was necessary since the solution of schizophrenia problems requires to rename the locally declared variables in the surfaces of loops (see below). Hence, we had to split the equation systems of the abbreviations so that renaming of only the surface part $E_{srfc}$ became possible. This, however, changes hash values, and moreover, allows one term to occur multiple times in different surfaces $E_{srfc}$ without being renamed.

To circumvent this later renaming and hence, the necessity to use multiple hash tables, the solution of the revised journal version [228] works differently: A function Renamings was used to compute all possible scopes before the
actual compilation and these substitutions are forwarded by the depth compilation. The compile algorithms of this chapter are a further improvement in that the renaming is done during the depth compilation and the generated incarnation levels are forwarded to the subsequent surface compilations.

Finally, these compilation procedures have been extended to handle modular (separate) compilation [54], which is essentially the compilation procedure described in this chapter.

In this section, we will solve this problem in that we transform the control and data flow to an equation system, so that the entire semantics is given in form of an equation system. For every location variable \( \ell \) of the program, there is exactly one initial and one transition equation of the forms \( \text{init}(\ell) = \text{srfc}_\ell \) and \( \text{next}(\ell) = \Omega_\ell \), respectively, where \( \text{srfc}_\ell \) is a boolean constant, and \( \Omega_\ell \) is an expression that only depends on the current values of the location, local, input, and output variables. As the control flow is given as such an equation system, it follows that the control flow is totally defined and deterministic: For all current values, we can evaluate the expressions \( \Omega_\ell \) to determine the next value for \( \ell \), so that we definitely can compute which of the locations are active at the next point of time.

For this reason, the results of this section additionally prove that the control flow of every Quartz program is deterministic. As Quartz has however nondeterministic statements, we have to note that this only holds, when we view the controllable input variables as inputs, since these variables nondeterministically choose their values.

The data flow will also be translated to an equation system where for every output and local variable (including their possible reincarnations), we generate either an invariant equation of the form \( x = \varphi \) or also initial and transition equations as described for the control flow. For memorized variables, and event variables with delayed actions, we must also define a so-called accompanying carrier variable to store the previous values if these are required. In contrast to the control flow, the data flow equations can suffer from causality cycles, since the dependency graph of the variables defined by the equations can be cyclic.

We will define a causality analysis for the equation systems that are generated from the Quartz statements. In principle, this causality analysis is a symbolic execution of the program that covers all possible inputs and all reachable states. Preliminarily unknown values are modeled with the special value \( \perp \) as already done in the previous chapter. We will outline the equivalence to the so-called ternary simulation of asynchronous circuits in Section ??.

A major problem for the translation into hardware circuits are \textit{schizophrenic statements}, in particular schizophrenic local declarations [42, 210, 233]. Recall that a statement is schizophrenic if some of its micro steps are executed more than once in a macro step. This may happen \textit{only if the statement be-}
longs to a loop body that is left and (re)entered at the same time (in the same macro step). If the scope of a local declaration is thereby left and (re)entered, then the compiler must carefully distinguish between different incarnations of local variables that exist at the same time, but in different scopes.

In the previous chapter, we already dealt with the reincarnations in that we used incarnation level functions $h$ to map the local variables to their current incarnation level. This means that we generated for every reincarnation of a variable $x$ a new variable of a different name $h(x)$, which is however known to be the $i$-th incarnation of $x$. In general, several copies may be necessary, which can (only) be achieved by nested abort and loop statements as shown by Gonthier's example in Figure 3.2 on page 60). We will use the same notation and the same incarnation level functions also in this chapter.

In this section, we therefore present an algorithm for the translation of Quartz programs to equation systems that can also be interpreted as hardware circuits. In a first step, we compute thereby directly the equation system of the control flow, while the data flow is computed in terms of guarded actions as done in the previous chapter. In contrast to the previous chapter, we then construct equations of the guarded actions, to that the entire result is an equation system that covers both the control and data flow of the program.

5.1 Revisiting Schizophrenia Problems

We have already seen in Section 3.2 that subtle problems may arise when local declarations are nested within loop statements. The problem is thereby that a local declaration can be left and re-entered within the same macro step. The micro steps of such a macro step must then refer to the right incarnation of the local variable, depending on whether they belong to the old or the new scope of the local declaration. Local declarations that yield different incarnations of a local variable at the same point of time are called schizophrenic in [42] (Chapter 12).

As an example, consider the program given on the left hand side of Figure 5.1. The right hand side of Figure 5.1 shows a the program where reincarnated surfaces are explicitly listed and where the reincarnated variables are decorated with their incarnation levels. Initially, we execute immediate assignments of different scopes of the initial surfaces of the local variables, so that we have $a=b=c=0$ when the control flow reaches the location $\ell$.

The further execution depends on the inputs $i$, $j$ and $k$. If both $j$ and $k$ hold, then we execute all assignments below the location $\ell$. If additionally $i$ holds, the control flow leaves the statement due to the abortion, otherwise it reenters the location $\ell$. If $j$ does not hold, but $k$ does, then the following assignments are executed:
module R1(event i,j,k, int x) {
    weak abort
    do {
        int a;
        a = 0;
        weak abort
        do {
            int b;
            b = a;
            weak abort
            do {
                int c;
                c = b;
                next(c) = f(a,b,c);
                ℓ:pause;
                x = a;
                next(a) = g(a,b,c);
                next(b) = h(a,b,c);
            } while(true)
        } when(k)
        } while(true)
    } when(j)
    } while(true)
} when(i)
}

module R1(event i,j,k, int x) {
    a = 0;
    b = a;
    c = b;
    next(c) = f(a,b,c);
    weak abort
    do {
        weak abort
        do {
            weak abort
            do {
                ℓ:pause;
                x = a;
                next(a) = g(a,b,c);
                next(b) = h(a,b,c);
                c(1) = b;
                next(c) = f(a,b,c(1));
            } while(true)
        } when(k);
        b(1) = a;
        c(2) = b(1);
        next(c) = f(a,b(1),c(2));
    } while(true)
    } when(j);
    a(1) = 0;
    b(2) = a(1);
    c(3) = b(2);
    next(c) = f(a(1),b(2),c(3));
} while(true)
} when(i)
}

Fig. 5.1. Local Declarations with Delayed Actions.

x = a;
next(a) = g(a,b,c);
next(b) = h(a,b,c);
c(1) = b;
next(c) = f(a,b,c(1));
b(1) = a;
c(2) = b(1);
next(c) = f(a,b(1),c(2));

Hence, up to three reincarnations of c exist in parallel to the depth value of c. Since in hardware circuits, every wire has exactly one value per clock cycle, a hardware synthesis must safely distinguish between the different reincarna-

5.1 Revisiting Schizophrenia Problems

This can be done as in the previous Chapter by using incarnation level functions that are updated during the compilation.

For software generation, one could implement the incarnations simply by shadowing the incarnations of the old scope. However, while this is reasonable for a single processor system, it seems to be a restricting condition for a multiprocessor approach that could work on different incarnations at the same time. Therefore, it is also reasonable to generate several copies of the locally declared variables according to the number of the possible ‘(re)enterings’.

Delayed actions add further difficulties to the reincarnation of locally declared variables: If a delayed action that changes the value of a locally declared variable is executed at termination time of the local declaration, then we have to disable its execution (at least when the local declaration is (re)entered at the same time). This must be done even if the reason for the termination is a weak abortion, since the scope is left, and therefore, this incarnation of the variable is no longer alive at the next point of time. If we (re)enter the local declaration at the same point of time, we must not transfer the delayed value to the new scope. For example, in Figure 5.1, at most one of the actions next(c) = f(a, b, c), next(c) = f(a, b, c\(^{(1)}\)), next(c) = f(a, b\(^{(1)}\), c\(^{(2)}\)), and next(c) = f(a\(^{(1)}\), b\(^{(2)}\), c\(^{(3)}\)) must be executed. We already handled this point in the definition of the surface and depth actions in that we explicitly disabled the guards of the corresponding actions. We will do the same in the hardware synthesis that is presented in the following.

Delayed delayed actions on local variables must also be disabled when the local declaration is instantaneously executed, since then the variable is no longer alive at the next point of time. Note that the disabling of delayed actions on local variables is a subtle issue. The following two examples show two possibly unexpected behaviors:

```plaintext
module P(int &y) { module Q(int &y) {
  weak suspend {
    int x;
    \ell: pause;
    next(x)=x+1;
    y=x;
  } when(true)
  }
  loop
  weak abort {
    int x;
    \ell: pause;
    next(x)=x+1;
    y=x+2;
  } when(true)
}
```

If the control flow has reached location \(\ell\) in module P, the local variable \(x\) has its initial value 0. Then, \(x\) is incremented, and \(y\) takes the current value of \(x\). As the suspension takes place, the control flow can not leave location \(\ell\) and will therefore be still at that location in all of the following points of time, so that the same behavior is infinitely often repeated. Hence, the behavior of module P is that \(y\) is zero for the first two points of time, and then enumerates all positive numbers.
If the control flow has reached location $\ell$ in module $Q$, then $x$ has value 0 and the abortion takes place. Since it is a weak abortion, the data actions are retained. Hence, $y$ is 2 at time 1, but the assignment $\text{next}(x)=x+1$ is not executed. This assignment is disabled not primarily because of the abortion, but because of the fact that at the next point of time the variable $x$ that should receive the value 1 is no longer alive. Hence, when the loop re-enters the local declaration, the same behavior is repeated. The assignment $\text{next}(x)=x+1$ is therefore dead code, and $y=x+2$ can be replaced with $y=2$.

Finally, as already explained in the construction of the data flow transition relation, we have to compute the preconditions of the surfaces that are generated by the loops. This is required to check which of the reincarnations really take place, and in particular, which of the reincarnations is the most recent one that will then transfer its value to the depth (in case of memorized variables). For example, if we would remove the assignment $\text{next}(c) = f(a,b,c)$ in the program of Figure 5.1, then there will only be an assignment $c=b$ in the surface of the local declaration of $c$, and there is no other assignment to $c$. If we rename all the occurrences of $c$ in the surfaces, then there is no assignment to $c$ at all. However, the assignments after the location $\ell$ (in the original program) read the value of $c$ in their right hand sides. Therefore, we have to select the right reincarnation $c^{(1)}$, $c^{(2)}$, and $c^{(3)}$ as the value of $c$ in the depth.

We must also disable delayed actions on local variables in those surfaces of local declarations that do not directly proceed to their depth. Note that the surface of a local declaration can be executed more than once (see Figure 5.1), but at most one of these surface instances can proceed to the depth without leaving the scope. Only delayed actions of this instance of the surface are executed. This is done by disabling the delayed actions on the local variables also in case that the local declaration is instantaneous.

### 5.2 Translation to Guarded Actions

In this section, we present the translation from Quartz statements to equivalent guarded actions in full detail. In order to correctly handle schizophrenia problems and compilation of preemption statements, this computation must be split into the computation of the surface and depth actions. For this reason, two functions CompSurface and CompDepth are described in the following two sections. We discuss the overall compilation and linking of separately compiled modules in the final subsection of this chapter.

In order to share common sub-expressions both functions CompSurface and CompDepth make use of an associative store that maps variables to boolean expressions. This associative store is updated by a function NewDef that works as follows: Given a boolean expression $\varphi$, the function NewDef($\varphi$) first checks whether there is already a variable $x_\varphi$ associated with $\varphi$. If this is the case, the corresponding variable $x_\varphi$ is returned. Otherwise, function
5.2 Translation to Guarded Actions

NewDef generates a new variable $x_\varphi$, associates this variable with expression $\varphi$ and returns this variable $x_\varphi$. Semantically, this means that we identify $x_\varphi$ with $\varphi$, and therefore we could essentially remove all occurrences of calls to function NewDef (this is just a matter of efficient memory use). However, experimental results have shown that sharing common sub-expressions is mandatory for reasonable efficiency of the compiler.

The compiler will typically first associate any control flow expression of the program with a new variable in the mentioned associative store, and will then use this variable for further computations instead of the original control flow expression so that the blow-up due to generation of multiple occurrences of this expression is limited to occurrences of variables.

It is therefore important to note that the compilation is done in a way that all control flow expressions are associated with variables. However, modular compilation breaks this rule in that we have to interrupt the compilation (since we cannot go on with the compilation of the called module), and simply assume that after that module has been compiled, the linker will be available to access as file where the results of its compilation are found. The linker will then associate these results with some variables that have to be already provided as placeholders by the compiler when compiling the calling module. For this reason, the functions CompSurface and CompDepth make use of an auxiliary function NewVar that simply introduces a new variable in the aforementioned associative store without yet associating it with a corresponding expression. The missing expression will then be later associated with this variable by the linker as explained in the final section of this chapter.

5.2.1 Computing Guarded Actions of the Surface

As already mentioned, function CompSurface is used to compute the guarded actions of the surface of a given statement $S$. To this end, two input conditions strt and prmt are provided which determine the starting mode of the considered statement. This starting mode is encoded as shown below:

<table>
<thead>
<tr>
<th>strt</th>
<th>prmt</th>
<th>starting mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>*</td>
<td>no start</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>weakly preempted start</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>start without preemption</td>
</tr>
</tbody>
</table>

Condition strt must hold in order to start the execution of the considered statement at all. It may be the case that the statement is already preempted at starting time which is indicated by condition prmt. Note that we only have to care about weak preemption, since strong preemption will lead to an invalid start condition. Hence, the mentioned preemption is always a weak one: A start without preemption executes all data flow actions of the statement, and enters the control flow locations, while a weakly preempted start only executes the data flow actions, but does not enter the control flow locations. The
theoretically possible fourth case, where only the control flow locations are entered, but no data flow actions are executed is not useful, and therefore this case does not occur.

Moreover, function \textit{CompSurface} requires a substitution \( \varrho \) to rename the local variables according to their reincarnation indices of this surface (as described in the previous section). This substitution is constructed in the compilation of the depth of a loop based on its current incarnation index. It is applied by function \textit{CompSurface} to every expression of the statement before using it to compute any results. A substitution is thereby implemented as a set of pairs \((x, \tau)\) denoting that variable \(x\) is mapped to expression \(\tau\), so that the empty set encodes the identity function. As the renaming \( \varrho \) is forwarded without changes, it is possible to store it in a global variable so that it will no longer be an argument of \textit{CompSurface}.

Using the starting conditions \(\texttt{strt}\) and \(\texttt{prmt}\), the renaming \(\varrho\) of the local variables of this surface, and the statement \(S\), function \textit{CompSurface} computes a tuple \((\texttt{inst}, \mathcal{D}_{\text{surf}}, \mathcal{G}_{\text{trf}}, \mathcal{M}_{\text{surf}}, \mathcal{C}_{\text{surf}})\) with the following meaning, where \(h_{\varrho}\) is the incarnation level function that corresponds with the substitution \(\varrho\) (correspondence means that \(h_{\varrho}(x) = \varrho(x)\) for all variables \(x\):

- \(\texttt{inst}\) is the condition that holds iff the renamed statement is currently instantaneous (i.e. \(\text{inst}_{h_{\varrho}}(S)\)),
- \(\mathcal{D}_{\text{surf}}\) is the set of guarded actions \((\gamma, \mathcal{A})\) of the data flow of the surface,
- \(\mathcal{G}_{\text{trf}}\) is the set of transfer actions \((x, (\gamma, \varrho(x)))\) that are used to transfer values from reincarnations \(\varrho(x)\) to the original local variable \(x\) in case \(\gamma\) is true,
- \(\mathcal{M}_{\text{surf}}\) is the set of module calls made in the surface (each one is a tuple \((\texttt{nameI}, \texttt{nameM}, x_{\text{inst}}, \texttt{strt}, \texttt{prmt}, [(\tau_1, \varphi_1), \ldots, (\tau_n, \varphi_n)]\) consisting of the instance name \(\texttt{nameI}\), the module’s name \(\texttt{nameM}\), the start context \(x_{\text{inst}}, \texttt{strt}, \texttt{prmt}\) of the module call, and the renamed argument expressions \((\tau_1, \ldots, \tau_n)\) endowed with possible disabling conditions \(\varphi_i\)), and
- \(\mathcal{C}_{\text{surf}}\) is the set of guarded actions of the control flow (these guarded actions are of the form \((\gamma, \text{next}(\ell) = \text{true})\) for a location \(\ell\)).

In the following, detailed explanations are given on the computation of these results. The compilation of actions is almost self-explaining: their executions are instantaneous, and their guard is simply the condition \(\texttt{strt}\). The renaming of actions is however a bit subtle, since we must only rename occurrences that refer to the current point of time. For its implementation, we use some auxiliary functions for left-hand side expressions:

- Function \texttt{BaseVar}(\lambda) computes for a left-hand side expression \(\lambda\) the base variable which is defined as follows:
  - \(\texttt{BaseVar}(x) := x\)
  - \(\texttt{BaseVar}(\lambda.\tau) := \texttt{BaseVar}(\lambda)\)
  - \(\texttt{BaseVar}(\lambda[\tau]) := \texttt{BaseVar}(\lambda)\)
function CompSurface(\(\varrho, strt, prmt, S\))

\[
\text{case } S \text{ of}
\]

\[
isAction(S): \quad \text{return} \ (true, \{(strt, RenameAct(\(\varrho, S\)))\}, \{\}, \{\});
\]

nothing:

\[
\ell: \text{pause}: \quad \text{return} \ (false, \{\}, \{\}, \{(strt \land \neg prmt, \text{next}(\ell) = true)\});
\]

if(\(\sigma\)) \(S_1\) else \(S_2\):

\[
\text{return} \ \text{ConditionalCompSurface}(\(\varrho, strt, prmt, \sigma, S_1, S_2\));
\]

\[
\{S_1; S_2\}: \quad \text{return} \ \text{SequenceCompSurface}(\(\varrho, strt, prmt, S_1, S_2\));
\]

\[
\{S_1 \parallel S_2\}: \quad \text{return} \ \text{ParallelCompSurface}(\(\varrho, strt, prmt, S_1, S_2\));
\]

\[
do \ S \ while(\(\sigma\)): \quad \text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S\));
\]

during \(S_1\) do \(S_2\):

\[
\text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S_1\));
\]

\[
\{\alpha \ x; S\}: \quad \text{return} \ \text{LocalDeclCompSurface}(\(\varrho, strt, prmt, \alpha, x, S\));
\]

abort \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S\));
\]

weak abort \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S\));
\]

immediate abort \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{AbortCompSurface}(\(\varrho, strt, prmt, false, \sigma, S\));
\]

weak immediate abort \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{AbortCompSurface}(\(\varrho, strt, prmt, true, \sigma, S\));
\]

suspend \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S\));
\]

weak suspend \(S\) when(\(\sigma\)):

\[
\text{return} \ \text{CompSurface}(\(\varrho, strt, prmt, S\));
\]

\[
\ell: \text{immediate suspend } S \text{ when(}\sigma\): \quad \text{return} \ \text{SuspendCompSurface}(\(\varrho, strt, prmt, \ell, false, \sigma, S\));
\]

\[
\ell: \text{weak immediate suspend } S \text{ when(}\sigma\): \quad \text{return} \ \text{SuspendCompSurface}(\(\varrho, strt, prmt, \ell, true, \sigma, S\));
\]

\[
namelm:nameM(argL): \quad \text{return} \ \text{ModuleCallCompSurface}(\(\varrho, strt, prmt, namelm, nameM, argL\));
\]

end

end

Fig. 5.2. Computing Guarded Actions of the Surface (Part I)
function ConditionalCompSurface(\(\rho, \text{strt}, \text{prmt}, \sigma, S_1, S_2\))
    \(\text{strt}_1 := \text{NewDef}(\text{strt} \land \neg g(\sigma))\);
    \(\text{strt}_2 := \text{NewDef}(\text{strt} \land g(\sigma))\);
    \((\text{inst}_1, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1) := \text{CompSurface}(\rho, \text{strt}_1, \text{prmt}, S_1)\);
    \((\text{inst}_2, D_{\text{strf}}^2, G_{\text{strf}}^2, M_{\text{strf}}^2, C_{\text{strf}}^2) := \text{CompSurface}(\rho, \text{strt}_2, \text{prmt}, S_2)\);
    \(\text{inst} := \text{NewDef}(\text{inst}_1 \land g(\sigma) \lor \text{inst}_2 \land \neg g(\sigma)) \land \text{inst}_1 \land \text{inst}_2)\);
    \(\text{return} (\text{inst}, D_{\text{strf}}^1 \cup D_{\text{strf}}^2, G_{\text{strf}}^1 \cup G_{\text{strf}}^2, M_{\text{strf}}^1 \cup M_{\text{strf}}^2, C_{\text{strf}}^1 \cup C_{\text{strf}}^2)\)
end

function SequenceCompSurface(\(\rho, \text{strt}, \text{prmt}, S_1, S_2\))
    \((\text{inst}_1, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1) := \text{CompSurface}(\rho, \text{strt}, \text{prmt}, S_1)\);
    \(\text{ strt}_2 := \text{NewDef}(\text{strt} \land \text{inst}_1)\);
    \((\text{inst}_2, D_{\text{strf}}^2, G_{\text{strf}}^2, M_{\text{strf}}^2, C_{\text{strf}}^2) := \text{CompSurface}(\rho, \text{strt}_2, \text{prmt}, S_2)\);
    \(\text{inst} := \text{NewDef}(\text{inst}_1 \land \text{inst}_2)\);
    \(\text{return} (\text{inst}, D_{\text{strf}}^1 \cup D_{\text{strf}}^2, G_{\text{strf}}^1 \cup G_{\text{strf}}^2, M_{\text{strf}}^1 \cup M_{\text{strf}}^2, C_{\text{strf}}^1 \cup C_{\text{strf}}^2)\)
end

function ParallelCompSurface(\(\rho, \text{strt}, \text{prmt}, S_1, S_2\))
    \((\text{inst}_1, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1) := \text{CompSurface}(\rho, \text{strt}, \text{prmt}, S_1)\);
    \((\text{inst}_2, D_{\text{strf}}^2, G_{\text{strf}}^2, M_{\text{strf}}^2, C_{\text{strf}}^2) := \text{CompSurface}(\rho, \text{strt}, \text{prmt}, S_2)\);
    \(\text{inst} := \text{NewDef}(\text{inst}_1 \land \text{inst}_2)\);
    \(\text{return} (\text{inst}, D_{\text{strf}}^1 \cup D_{\text{strf}}^2, G_{\text{strf}}^1 \cup G_{\text{strf}}^2, M_{\text{strf}}^1 \cup M_{\text{strf}}^2, C_{\text{strf}}^1 \cup C_{\text{strf}}^2)\)
end

function AbortCompSurface(\(\rho, \text{strt}, \text{prmt}, wk, \sigma, S\))
    \(\text{strt}_1 := \text{if} \ wk \text{ then } \text{strt} \text{ else } \text{NewDef}(\text{strt} \land \neg g(\sigma)) \text{ end;}
    \text{prmt}_1 := \text{NewDef}(\text{prmt} \lor g(\sigma))\);
    \((\text{inst}_1, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1) := \text{CompSurface}(\rho, \text{strt}_1, \text{prmt}_1, S)\);
    \(\text{inst} := \text{NewDef}(\text{inst} \lor g(\sigma))\);
    \(\text{return} (\text{inst}, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1)\)
end

function SuspendCompSurface(\(\rho, \text{strt}, \text{prmt}, \ell, wk, \sigma, S\))
    \(C_{\text{strf}}^1 := \{(\text{strt} \land \neg \text{prmt} \land g(\sigma), \text{next}(\ell) = \text{true})\};
    \(\text{strt}_1 := \text{if} \ wk \text{ then } \\text{strt} \text{ else } \text{NewDef}(\text{strt} \land \neg g(\sigma)) \text{ end;}
    \text{prmt}_1 := \text{NewDef}(\text{prmt} \lor g(\sigma))\);
    \((\text{inst}_1, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1) := \text{CompSurface}(\rho, \text{strt}_1, \text{prmt}_1, S)\);
    \(\text{inst} := \text{NewDef}(\text{inst} \land \neg g(\sigma))\);
    \(\text{return} (\text{inst}, D_{\text{strf}}^1, G_{\text{strf}}^1, M_{\text{strf}}^1, C_{\text{strf}}^1, C_{\text{strf}}^0)\)
end

Fig. 5.3. Computing Guarded Actions of the Surface (Part II)
function LocalDeclCompSurface(\(\varrho, \text{str}, \text{prmt}, \alpha, x, S\))

function DLV(\(\xi, \lambda, \gamma\))

\[
\gamma := \begin{cases} 
\text{false} & \text{if } \text{isLhs}(\lambda) \land \text{BaseVar}(\lambda) = x \text{ then return } \gamma \land \xi \land \gamma; \\
\end{cases}
\]

end

function DisableDelayedLV(\(\xi, \gamma, A\))

\[
\text{case } A \text{ of} \begin{cases} 
\text{next}(\lambda) = \tau: \text{return } (\text{DLV}(\lambda, \gamma), \text{next}(\lambda) = \tau); \\
\end{cases}
\]

else return \((\gamma, A)\)

end

function DisableDelayedLMC(\(\xi, (\text{nameI}, \text{nameM}, x_{\text{inst}}, \text{str}, \text{prmt}, \text{argL})\))

\[
[\{(\tau_1, \varphi_1), \ldots, (\tau_n, \varphi_n)\}] := \text{argL};
\]

\[
\text{argL} := [(\tau_1, \text{DLV}(\xi, \tau_1, \varphi_1)), \ldots, (\tau_n, \text{DLV}(\xi, \tau_n, \varphi_n))];
\]

return \((\text{nameI}, \text{nameM}, x_{\text{inst}}, \text{str}, \text{prmt}, \text{argL}');\)

end

\[
(\text{inst}, \mathcal{D}^{\text{nf}}, \mathcal{G}^{\text{nf}}, \mathcal{M}^{\text{nf}}, \mathcal{C}^{\text{nf}}) := \text{CompSurface}(\varrho, \text{str}, \text{prmt}, S);
\]

\[
\xi := \neg(\text{prmt} \lor \text{inst});
\]

\[
\mathcal{G}^{\text{nf}} := \{(x, (\text{str} \land \neg \text{prmt}, \varrho(x)))\} \cup \mathcal{G}^{\text{nf}};
\]

\[
\mathcal{D}^{\text{nf}} := \text{DisableDelayedLV}(\xi, \gamma, A) \mid (\gamma, A) \in \mathcal{D}^{\text{nf}};
\]

\[
\mathcal{M}^{\text{nf}} := \{\text{DisableDelayedLMC}(\xi, m) \mid m \in \mathcal{M}^{\text{nf}}\};
\]

return \((\text{inst}, \mathcal{D}^{\text{nf}}, \mathcal{G}^{\text{nf}}, \mathcal{M}^{\text{nf}}, \mathcal{C}^{\text{nf}})\)

end

function ModuleCallCompSurface(\(\varrho, \text{str}, \text{prmt}, \text{nameI}, \text{nameM}, \text{argL}\))

\[
x_{\text{inst}} := \text{NewVar}();
\]

\[
[\tau_1, \ldots, \tau_n] := \text{argL};
\]

\[
\text{argL} := [(\varrho(\tau_1), \text{true}), \ldots, (\varrho(\tau_n), \text{true})];
\]

\[
m := (\varrho(\text{nameI}), \text{nameM}, x_{\text{inst}}, \text{str}, \text{prmt}, \text{argL}');
\]

return \((x_{\text{inst}}, \{\}, \{\}, \{m\}, \{\})\)

end

Fig. 5.4. Computing Guarded Actions of the Surface (Part III)

- Function RenameLhs(\(\varrho, \lambda\)) applies the substitution \(\varrho\) to all sub-expressions of a left-hand side expression \(\lambda\) that are not the target variable BaseVar(\(\lambda\)) (since that occurrence refers to the depth):
  - RenameLhs(\(\varrho, x\)) := \(x\)
  - RenameLhs(\(\varrho, \lambda.\tau\)) := RenameLhs(\(\varrho, \lambda\), \(\varrho(\tau)\))
  - RenameLhs(\(\varrho, \lambda[\tau]\)) := RenameLhs(\(\varrho, \lambda\)[\(\varrho(\tau)\)]

- Finally, the renaming RenameAct(\(\varrho, \alpha\)) of an action \(\alpha\) with a substitution \(\varrho\) excludes the base variable in left-hand side expressions:
  - RenameAct(\(\varrho, \text{assert}(\sigma)\)) := \text{assert}(\varrho(\sigma))
  - RenameAct(\(\varrho, \text{assume}(\sigma)\)) := \text{assume}(\varrho(\sigma))
  - RenameAct(\(\varrho, \lambda = \tau\)) := \(\varrho(\lambda) = \varrho(\tau)\)
The compilation of the nothing statement is trivial. The compilation of $\ell : \text{pause}$ is also not difficult, and we only have to discuss the returned guarded action $(\text{strt} \land \neg \text{prmt}, \text{next}(\ell) = \text{true})$ of the control flow. If the statement is not be started or started under an immediate strong preemption, condition $\text{strt}$ would not hold, and therefore the guard $\text{strt} \land \neg \text{prmt}$ would also not hold. If the statement would be started under a weak preemption, we would have $\text{strt} \land \text{prmt}$, so that the guard $\text{strt} \land \neg \text{prmt}$ would also not hold. Finally, if the statement is started without preemption, we have $\text{strt} \land \neg \text{prmt}$, which is the guard of the action that therefore sets the control flow location $\ell$ in the next macro step.

Considering conditional statements $\text{if}(\sigma) \ S_1 \ \text{else} \ S_2$, it should be remarked that we return an expression for the instantaneous condition that allows one to deduce a value even though the value of the branch condition $\sigma$ might not be known (see McCarthy’s parallel if-then-else [64] and the discussion on multi-valued logics used in causality analysis in the previous chapter). Clearly, we could alternatively return an alternative expression that would lead to other definitions of constructive programs [229, 230].

The surface of a sequence $\{S_1; S_2\}$ includes the surface of $S_1$ and in case $S_1$ is instantaneous, also the surface of $S_2$. In case the instantaneousity of $S_2$ is a dynamic condition, i.e. neither one of the constants true and false, we have to compute copies of the surface of $S_2$ (one in the surface and another one in the depth). Note that even though these different surfaces have different starting conditions, they can be both started within the same macro step (see discussion on page 177).

The compilation of parallel statements $\{S_1 \parallel S_2\}$ is trivial; there are no further comments. Perhaps, it is interesting to see that the compilation returns exactly the same result as for a sequence $\{S_1; S_2\}$ in case $\text{inst}_1 = \text{true}$ holds.

The surface of do $S$ while(\sigma) is just the surface of the loop body $S$. Note that by definition of the semantics, $S$ must never be instantaneous, so that we might demand that the expression $\text{inst}$ generated must be reducible to the constant false. We might also generate an assertion for this purpose.

The surface of during $S_1$ do $S_2$ is just the surface of $S_1$. Recall that $S_2$ is only executed after the control flow has entered $S_1$ and $S_1$ does not terminate at the same point of time.

Surfaces of delayed preemption statements are just the surfaces of their body statements, since these statements have no effect when they are started. Immediate preemption statements, however, modify the starting conditions $\text{strt}$ and $\text{prmt}$ by adding the additional preemption condition. In case of weak preemption statements, this is left out for $\text{strt}$ so that a weak preemptive start is still possible. For strong preemption statements, however, also the $\text{strt}$ condition must be invalidated in case the preemption condition holds to avoid any action of the body statement.
5.2 Translation to Guarded Actions

module R2(event ?i, ?j, !x) {
    do {
        int a;
        a = 0;
        weak abort
do {
        int b;
        b = a;
        weak abort
do {
        int c;
        c = b;
        w1: pause;
        x = c;
    } while(true)
when(i)
} while(true)
when(j)
} while(true)
}

module R2(event ?i, ?j, !x) {
    int a, a\(3\), b, b\(2\), b\(3\), c, c\(1\), c\(2\), c\(3\);
a = 0;
b = a;
c = b;
do {
    weak abort
do {
    weak abort
do {
    w1: pause;
    x = c;
c\(1\) = b;
} while(true)
when(i);
} while(true)
when(j);
} while(true)
}

Fig. 5.5. A module with the need of transfer actions.

The compilation of local variable declarations \{α \ x; S\} clearly needs more attention even though most of the task is already done by the incarnation indices and the renaming by the substitution \(\varrho\). However, there are two important tasks left:

- **Computing Transfer Actions:** As described in the previous section, a local variable might have several reincarnations, depending on the number of loops that enclose its declaration. If a local variable has storage mode ‘memorized’, and if neither a delayed assignment is enabled when entering the local declaration nor an immediate assignment is enabled right after entering it, then the local variable should store its previous value. However, this previous value is the value of one of its reincarnations, and therefore the compiler must select the right reincarnation.

As an example, consider the module shown in Figure 5.5, and assume the control flow is currently at location w1 and the current inputs i and j are true. Similar to the example of the previous section, we execute the assignments right after w1, the innermost loop restarts its body, is then weakly aborted, so that the assignments right after weak abort . . . when(i) are executed. Then, the second loop restarts its body, is then also weakly
aborted, so that the assignments right after `weak abort ... when(j)` are executed. After this, the outermost loop is restarted, so that location `w1` is reached again. Now, note that the local variables all have storage mode 'memorized', so that the assignment `x = c` must know the previous value of variable `c`. As there is no immediate assignment to `c` after the initial point of time, and no delayed assignments to `c` at all, we have to select the right reincarnation of `c` to determine the 'previous value of `c`'.

In general, the transfer value has to be taken from the reincarnation whose surface is started without preemption. All other surfaces are either not started or are started with a weak preemption, so that the control flow will not reach a location within the local declaration without looping over further surfaces. For this reason, `CompSurface` returns transfer actions $G^{trf}$ that consist of tuples $(x, (\text{strt} \land \neg \text{prmt}, \varphi(x)))$ with the meaning that local variable `x` may take its previous value from reincarnation $\varphi(x)$. Note that each local declaration generates exactly one transfer action per surface. The different surfaces are ordered by the nesting of the loops that generate them, and this ordering is maintained in the compilation of the depth in the absence actions as described there.

Note that during compilation, the transfer actions must be kept separately from the other guarded actions, since transfer actions have lower priority than the other actions. After compilation (and linking), we can weaken their guards accordingly to avoid pseudo write conflicts with the guarded actions determined by the data flow of the program.

```plaintext
module M(event bool ?i,...) {
  w1: pause;
  w2: weak immediate suspend {
    nat x;
    next(x) = 5;
    w3: pause;
    ...
  } when(i)
}
```

Fig. 5.6. Immediate suspension aborts delayed assignments to local variables.

• **Disabling Delayed Assignments to Local Variables:** As already mentioned, delayed assignments to local variables must not be executed at the point of time where the local variable declaration is left and re-entered. The surface of a local declaration can be entered and left for three reasons:
  – The local declaration is currently instantaneous and is therefore immediately left when started. In this case, a delayed assignment to its local
variable is useless. Even though it might not be harmful, we conservatively disable delayed assignments in this case.

- An abortion statement that encloses the local declaration immediately preempts its execution. It is thereby irrelevant whether the abortion is a weak or a strong one, so that in case of weak abortions, we have to additionally disable delayed assignments to local variables.

- Surprisingly, suspension statements can have a similar effect as demonstrated by the module shown in Figure 5.6: If the control starts from location \( w_1 \) when input \( i \) holds, then the delayed assignment to \( x \) must not be executed, since the scope of the local declaration of \( x \) is immediately left, so that the control flow will rest at location \( w_2 \) at the end of the macro step. From there, it will reopen another scope of \( x \) at the next macro step. Hence, also an immediate suspension of a local declaration requires to disable the delayed assignments that were otherwise executed in its surface.

Hence, we have to disable all delayed assignments to local variables in case the surface is preempted or instantaneous. For this reason, the implementation shown in Figure 5.4 adds the condition \( \xi := \neg (\text{prmt} \lor \text{inst}) \) to all guards of these actions.

Separate compilation of modules imposes another difficulty related with local variables: When a module is compiled, a delayed action might refer to an output variable, but when this module called in another module, the output variable of the called module might be associated with a local variable of the calling module. For this reason, we also have to disable delayed assignments to output variables if those are associated with local variables in a module call.

For this reason, the compiler stores for each argument expression \( \tau_i \) of a module call \( \text{nameI} : \text{nameM}(\tau_1, \ldots, \tau_n) \) a condition \( \phi_i \). If \( \tau_i \) is a left-hand side expression that refers to a local variable of the calling module, and the corresponding argument was declared as output of the called module, the linker will finally add \( \phi_i \) as an additional conjunct to the guards of all delayed actions to that output variable. Note that in a separate compilation, the compiler has no means to handle this problem, since it cannot foresee the later uses of a module that is currently compiled.

Finally, the compilation of surfaces of module calls is simple: The compiler just saves the conditions \( \text{strt} \) and \( \text{prmt} \) so that the linker can later associate them with variables used for the separate compilation (see the section on the linker). For a similar purpose, the compiler generates a variable \( x_{\text{inst}} \) for the instantaneity condition \( \text{inst} \) that is used instead of the condition that the linker will later on receive when the called module has been compiled. If the compilation of the called module has been done, the linker can finally associate \( x_{\text{inst}} \) with the condition \( \text{inst} \).

Besides storing the conditions \( \text{strt} \) and \( \text{prmt} \), and generating a variable \( x_{\text{inst}} \) to be associated with the instantaneity condition of the called module,
we add disabling conditions to each argument expression $\tau_i$ of the module call. The need of these disabling conditions has been explained above: since we later on might have to add disabling conditions as conjuncts, so that the initial conditions are simply true. Further compilation of a surrounding local declaration might add other conditions for disabling, so that the linker can finally correct the guards of delayed actions to local variables.

The correctness result of function $\text{CompSurface}$ given in Figures 5.2-5.4 is formally stated in the following theorem:

**Theorem 5.1 (Compiling the Surface).** Given a Quartz statement $S$, a substitution $\varrho$ that corresponds to the incarnation level function $\varrho_{\varrho}$, a starting condition $\text{strt}$, and a preemption condition $\text{prmt}$, the call $\text{CompSurface}(\varrho, \text{strt}, \text{prmt}, S)$ computes a tuple $(\text{inst}, D_{\text{sr}}f_{\varrho}, G_{\text{tr}}f, M_{\text{sr}}f_{\varrho}, C_{\text{sr}}f_{\varrho})$ with the following meaning (taking the associations of variables into account):

- $\text{inst} = \text{inst}_{\varrho_{\varrho}}(S)$
- $D_{\text{sr}}f_{\varrho} = \text{ActSurf}_{\varrho_{\varrho}}(\text{strt}, S)$, i.e., the set of guarded actions of the data flow in the surface of $S$
- $G_{\text{tr}}f$ is a set of tuples $(x, (\gamma, \varrho(x)))$ called transfer actions which mean that the local variable $x$ must take its default value from its incarnation $\varrho(x)$ if the condition $\gamma$ holds (this default value must be taken only for memorized local variables $x$ after entering their scope in case no delayed action has been executed while entering the scope and no immediate assignment is executed right after entering the scope),
- $M_{\text{sr}}f_{\varrho}$ is a set of tuples $(\text{name}_I, \text{name}_M, x_{\text{inst}}, \text{strt}, \text{prmt}, \text{arg}_L')$ that correspond to module calls that can be made in the surface of $S$ with the following components:
  - $\text{name}_I$ is the name of the instance of the module call,
  - $\text{name}_M$ is the name of the called module,
  - $x_{\text{inst}}$ is the variable that is used as placeholder of the instantaneity condition $\text{inst}$ of the module and will be associated with that condition once it is known,
  - $\text{strt}$ and $\text{prmt}$ are the start conditions of the module call,
  - and $\text{arg}_L$ is a list of pairs $(\tau_i, \phi_i)$ where $(\tau_1, \ldots, \tau_n)$ are the renamed expressions used in the module call and $\phi_i$ are additional disabling conditions to be added to the guards of variables that are locally declared.
- $C_{\text{sr}}f_{\varrho}$ are the guarded actions of the control flow which is a possibly empty set of guarded actions of the form $(\gamma, \text{next}(\ell) = \text{true})$. If $C_{\text{sr}}f_{\varrho} = \{}$, then $S$ has no locations, $\text{enter}_{\varrho_{\varrho}}(S)$ is false, and therefore $\text{inst} = \text{true}$. Otherwise, the following formula is valid:

$$
\text{enter}_{\varrho_{\varrho}}(S) \leftrightarrow \bigwedge_{(\gamma, \text{next}(\ell) = \text{true}) \in C_{\text{sr}}f_{\varrho}} \gamma \rightarrow \text{next}(\ell)
$$

Hence, $\text{CompSurface}(\varrho, \text{strt}, \text{prmt}, S)$ computes the surface actions as well as the control flow predicates $\text{inst}_{\varrho_{\varrho}}(S)$ and $\text{enter}_{\varrho_{\varrho}}(S)$ that determine the control
flow at entering the surface. In contrast to the control flow predicates of the
previous Chapter, \( \text{enter}_R(S) \) is now given in the form of guarded actions.

### 5.2.2 Computing Guarded Actions of the Depth

Similar to function \( \text{CompSurface} \), we define in this subsection a function
\( \text{CompDepth} \) to compute the guarded actions of the depth of a statement. These
are the guarded actions that are executed after a statement has already been
entered by the control flow. Again, we have to provide further arguments for
this function, and again, the result of this function contains more than just
the guarded actions of the depth of \( S \).

Since a statement may be aborted or suspended due to surrounding pre-
emption statements, the computation of the guarded actions of the depth has
to take care of potential preemption statements. To this end, we distinguish
five different execution modes that are encoded by three boolean conditions
\( \text{susp} \) (weak or strong suspension), \( \text{abrt} \) (weak or strong abortion), \( \text{strg} \) (strong
preemption) that are given as arguments to function \( \text{CompDepth} \):

<table>
<thead>
<tr>
<th>( \text{susp} )</th>
<th>( \text{abrt} )</th>
<th>( \text{strg} )</th>
<th>execution mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>normal execution</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>weak abortion</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>strong abortion</td>
</tr>
<tr>
<td>true</td>
<td>*</td>
<td>false</td>
<td>weak suspension</td>
</tr>
<tr>
<td>true</td>
<td>*</td>
<td>true</td>
<td>strong suspension</td>
</tr>
</tbody>
</table>

Note that we do not have to distinguish between immediate or delayed pre-
emption statements, since for compiling the depth, we assume that the con-
trol flow has already entered the statement. Hence, immediate and delayed
preemption statement are compiled in exactly the same way (their difference
only shows up at starting time which is handled by \( \text{CompSurface} \)).

Note that suspension has a higher priority than abortion since the state-
ment is already suspended if \( \text{susp} \) holds while \( \text{abrt} \land \neg \text{susp} \) is required for an
abortion. This choice is arbitrary; we could alternatively give abortion state-
ments a higher priority or could even replace the don’t care * by false in the
above table. However, the latter would require more complex expressions in
the compiler, and the higher priority of abortion would require more complex
expressions for abortion contexts. Since we expect abortion statements to ap-
pear more often than suspension, the above encoding seems to be preferable.

We moreover have the invariant \( \text{strg} \rightarrow \text{abrt} \lor \text{susp} \), which excludes the re-
mainning case. In earlier versions of the translation procedure like [227, 228],
the input \( \text{strg} \) has not been used. This condition is used to define the start
conditions \( \text{strt} \) and \( \text{prmt} \) for the calls to \( \text{CompSurface} \) made by \( \text{CompDepth} \).
The earlier versions of the translation procedure ignored strong preemption
conditions, and added an additional conjunct to the guarded actions of the
depth of preemption statements in and additional post-processing step after
their computation. If Quartz modules should be compiled separately (as done here), this is also possible (and still be done for disabling delayed assignments to local variables), but we prefer to avoid this in case of preemption statements. In a separate compilation, one either has to add the additional guards by the later linking phase or to provide an explicit condition `strg` as done here (where that condition is stored).

In addition to the conditions `susp`, `abrt` and `strg`, function `CompDepth` also receives an integer \( h \) that denotes the current height of the incarnation level. Each time a loop is traversed, this incarnation level is increased (as required by the operational semantics to distinguish between different reincarnations of local variables). Clearly, the incarnation height \( h \) is used to determine a renaming \( \varrho \) to be used for calls to `CompSurface`.

Given an integer \( h \), and preemption conditions `susp`, `abrt` and `strg`, function `CompDepth` will return a tuple \((L, \text{insd}, \text{term}, D^\text{depth}, G^\text{abs}, M^\text{src}, M^\text{depth}, C^\text{depth})\) with:

- \( L \) is the set of local declarations inside \( S \)
- \( \text{insd} \) is equivalent to \( \text{in}(S) \)
- \( \text{term} \) is equivalent to \( \text{term}(S) \)
- \( D^\text{depth} \) is the set of guarded actions of the data flow of the depth of \( S \)
- \( G^\text{abs} \) is the set of absence reactions for the locally declared variables (cf. compilation of local declarations),
- \( M^\text{src} \) is the set of module calls to surfaces of other modules contained in the depth (cf. compilation of sequences, loops, immediate suspension, and during),
- \( M^\text{depth} \) is the set of module calls to depths of other modules inside \( S \),
- \( C^\text{depth} \) is the set of guarded actions of the control flow of the depth of \( S \)

The computation of the control flow predicates `insd` and `term` is thereby clear: We simply follow the recursive definition of these predicates as given in the previous chapter. The computation of the contained local declarations \( L \) is also clear: We simply add each local declaration that is traversed (cf. compilation of local declarations). The computation of the remaining results is discussed in the following for the different kinds of statements along the code of the `CompDepth` shown in Figures 5.7-5.10.

The depth of actions is the same as the depth of `nothing`: Since the control flow cannot rest inside these statements, there can be no resumption of the execution from within these statements, so that all entries are empty. Similarly, the resumption of the execution from a \( \ell : \text{pause} \) statement neither execute any actions nor does it call other modules. There is one guarded action of the control flow that sets \( \ell \) via a delayed assignment in case `susp \land \ell` holds, i.e., in case the control flow is currently at location \( \ell \) and a suspension takes place that freezes the control flow at the current location.

Compiling the depth of a conditional \( \text{if}(\sigma) \ S_1 \ 	ext{else} \ S_2 \) is similar to the compilation of the depth of a parallel statement \( \{S_1 \parallel S_2\} \): The depths of the
function CompDepth(h, susp, abrt, strg, S)
  case S of
    isAction(S):
      return ([], [], false, false, [], [], [], [])
    nothing:
      return ([], [], false, false, [], [], [], [])
    \ell: pause:
      return ([], [], \ell, \ell, [], [], [], {[susp \land \ell, next(\ell) = true]})
    if(\sigma) S_1 else S_2:
      return ConditionalCompDepth(h, susp, abrt, strg, \sigma, S_1, S_2)
      \{S_1; S_2\}:
      return SequenceCompDepth(h, susp, abrt, strg, S_1, S_2)
      \{S_1 || S_2\}:
      return ParallelCompDepth(h, susp, abrt, strg, S_1, S_2)
    do S while(\sigma):
      return DoWhileCompDepth(h, susp, abrt, strg, \sigma, S)
      \{\alpha; x; S\}:
      return LocalDeclCompDepth(h, susp, abrt, strg, \alpha, x, S)
    during S_1 do S_2:
      (L, insd, term, \mathcal{T}^{\text{depth}}, G^{\text{abs}}, M^{\text{src}}, M^{\text{depth}}, C^{\text{depth}})
      := CompDepth(h, susp, abrt, strg, S_1);
      \text{strg} := \text{NewDef}(\text{insd} \land \neg \text{term} \land \neg \text{strg});
      \text{prmt} := \text{NewDef}(\text{susp} \lor \text{abrt});
      \text{(insd, } G^{\text{src}}, M^{\text{src}}, C^{\text{src}} := \text{CompSurface}(\{\}. strt, prmt, S_2);
      \text{return } (L, \text{insd}, \text{term}, \mathcal{T}^{\text{depth}}, G^{\text{abs}}, M^{\text{src}} \cup M^{\text{src}}, M^{\text{depth}}, C^{\text{depth}})
    abort S when(\sigma):
      return AbortCompDepth(h, susp, abrt, strg, \sigma, S, false)
    immediate abort S when(\sigma):
      return AbortCompDepth(h, susp, abrt, strg, \sigma, S, false)
    weak abort S when(\sigma):
      return AbortCompDepth(h, susp, abrt, strg, \sigma, S, false)
    weak immediate abort S when(\sigma):
      return AbortCompDepth(h, susp, abrt, strg, \sigma, S, true)
    suspend S when(\sigma):
      return SuspendCompDepth(h, susp, abrt, strg, \sigma, S, false)
    weak suspend S when(\sigma):
      return SuspendCompDepth(h, susp, abrt, strg, \sigma, S, true)
    \ell: immediate suspend S when(\sigma):
      return SuspendCompDepth(h, susp, abrt, strg, \sigma, {\ell: pause; S}, false)
    \ell: weak immediate suspend S when(\sigma):
      return SuspendCompDepth(h, susp, abrt, strg, \sigma, {\ell: pause; S}, true)
    \text{name} = \text{nameM} argL):
      return ModuleCallCompDepth(h, susp, abrt, strg, \text{nameM}, \text{nameM}. argL)
  end
end

Fig. 5.7. Computing Guarded Actions of the Depth (Part I)
function ConditionalCompDepth\( (h, susp, abrt, strg, \sigma, S_1, S_2) \)
\( (L_1, \text{insd}_1, \text{term}_1, D_1^{dpth}, G_1^{abs}, M_1^{surf}, M_1^{dpth}, c_1^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_1); \)
\( (L_2, \text{insd}_2, \text{term}_2, D_2^{dpth}, G_2^{abs}, M_2^{surf}, M_2^{dpth}, c_2^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_2); \)
\( L := L_1 \cup L_2; \)
\( \text{insd} := \text{NewDef}(\text{insd}_1 \lor \text{insd}_2); \)
\( \text{term} := \text{NewDef}(\text{term}_1 \lor \text{term}_2); \)
\( D^{dpth} := D_1^{dpth} \cup D_2^{dpth}; \)
\( G^{abs} := G_1^{abs} \cup G_2^{abs}; \)
\( M^{surf} := M_1^{surf} \cup M_2^{surf}; \)
\( M^{dpth} := M_1^{dpth} \cup M_2^{dpth}; \)
\( c^{dpth} := c_1^{dpth} \cup c_2^{dpth}; \)
\( \text{return} \ (L, \text{insd}, \text{term}, D^{dpth}, G^{abs}, M^{surf}, M^{dpth}, c^{dpth}) \)
end

function SequenceCompDepth\( (h, susp, abrt, strg, S_1, S_2) \)
\( (L_1, \text{insd}_1, \text{term}_1, D_1^{dpth}, G_1^{abs}, M_1^{surf}, M_1^{dpth}, c_1^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_1); \)
\( (L_2, \text{insd}_2, \text{term}_2, D_2^{dpth}, G_2^{abs}, M_2^{surf}, M_2^{dpth}, c_2^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_2); \)
\( \text{strt}_2 := \text{NewDef}(\text{term}_1 \land \neg \text{strg}); \)
\( \text{prmt}_2 := \text{NewDef}(\text{ susp} \lor \text{ abrt}); \)
\( (L_2, \text{insd}_2, D_2^{dpth}, G_2^{abs}, M_2^{surf}, C^{dpth}) := \text{CompSurface}(\{\}, \text{strt}_2, \text{prmt}_2, S_2); \)
\( L := L_1 \cup L_2; \)
\( \text{insd} := \text{NewDef}(\text{insd}_1 \lor \text{insd}_2); \)
\( \text{term} := \text{NewDef}(\text{term}_1 \land \text{insd}_2 \lor \text{term}_2); \)
\( D^{dpth} := D_1^{dpth} \cup D_2^{surf} \cup D_2^{dpth}; \)
\( G^{abs} := G_1^{abs} \cup G_2^{abs}; \)
\( M^{surf} := M_1^{surf} \cup M_2^{surf} \cup M_2^{surf}; \)
\( M^{dpth} := M_1^{dpth} \cup M_2^{dpth}; \)
\( c^{dpth} := c_1^{dpth} \cup C_2^{dpth}; \)
\( \text{return} \ (L, \text{insd}, \text{term}, D^{dpth}, G^{abs}, M^{surf}, M^{dpth}, c^{dpth}) \)
end

function ParallelCompDepth\( (h, susp, abrt, strg, S_1, S_2) \)
\( (L_1, \text{insd}_1, \text{term}_1, D_1^{dpth}, G_1^{abs}, M_1^{surf}, M_1^{dpth}, c_1^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_1); \)
\( (L_2, \text{insd}_2, \text{term}_2, D_2^{dpth}, G_2^{abs}, M_2^{surf}, M_2^{dpth}, c_2^{dpth}) := \text{CompDepth}(h, susp, abrt, strg, S_2); \)
\( L := L_1 \cup L_2; \)
\( \text{insd} := \text{NewDef}(\text{insd}_1 \lor \text{insd}_2); \)
\( \text{term} := \text{NewDef}(\text{term}_1 \land \neg \text{insd}_2 \lor \text{term}_2 \land \neg \text{insd}_1 \lor \text{term}_1 \land \text{term}_2); \)
\( D^{dpth} := D_1^{dpth} \cup D_2^{dpth}; \)
\( G^{abs} := G_1^{abs} \cup G_2^{abs}; \)
\( M^{surf} := M_1^{surf} \cup M_2^{surf}; \)
\( M^{dpth} := M_1^{dpth} \cup M_2^{dpth}; \)
\( c^{dpth} := c_1^{dpth} \cup C_2^{dpth}; \)
\( \text{return} \ (L, \text{insd}, \text{term}, D^{dpth}, G^{abs}, M^{surf}, M^{dpth}, c^{dpth}) \)
end

Fig. 5.8. Computing Guarded Actions of the Depth (Part II)
two substatements $S_1$ and $S_2$ is computed recursively with the same arguments $h$, $susp$, $abrt$, and $strg$, and the unions of the obtained sets of guarded actions and module calls are computed. The only difference between these statements is the definition of the termination condition.

The depth of a sequence statement $\{S_1; S_2\}$ is more complicated: The control flow can either rest somewhere inside $S_1$ or inside $S_2$ (but not in both substatements). If the control flow is resumed from within $S_1$, we simply compute its depth with the arguments $h$, $susp$, $abrt$, and $strg$ used for the call to compute the sequence $\{S_1; S_2\}$. If the control flow moves from $S_1$ to $S_2$, it will first execute the surface of $S_2$ and then its depth. Hence, a computation of the surface of $S_2$ follows. To this end, the termination condition $term_1$ obtained
function LocalDeclCompDepth(h, susp, abrt, strg, α, x, S)
function DLV(ξ, λ, γ)
    γ := if isLhs(ξ) & BaseVar(λ) = x then return γ ∧ ξ else γ;
end
function DisableDelayedLV(ξ, γ, A)
    case A of
        next(λ) = τ: return (DLV(λ, γ), next(λ) = τ);
    else return (γ, A)
end
end
function DisableDelayedLMCS(ξ, (nameI, nameM, x_inst, strg, prmt, argL))
    [[(τ₁, φ₃), ..., (τₙ, φₙ)] := argL;
    argL' := [(τ₁, DLV(ξ, τ₃), φ₃), ..., (τₙ, DLV(ξ, τₙ, φₙ))];
    return (nameI, nameM, x_inst, strg, prmt, argL, argL');
end
function DisableDelayedLMCD(ξ, (nameI, nameM, x_ins, x_term, susp, abrt, strg, argL))
    [[(τ₁, φ₃), ..., (τₙ, φₙ)] := argL;
    argL' := [(τ₁, DLV(ξ, τ₃), φ₃), ..., (τₙ, DLV(ξ, τₙ, φₙ))];
    return (nameI, nameM, x_ins, x_term, susp, abrt, strg, argL');
end
end

function ModuleCallCompDepth(h, susp, abrt, strg, nameI, nameM, argL)
    x_ins := NewVar();
    x_term := NewVar();
    [τ₁, ..., τₙ] := argL;
    argL' := [(g(τ₁), true), ..., (g(τₙ), true)];
    m := (nameI, nameM, x_ins, x_term, susp, abrt, strg, argL');
    return ((), x_ins, x_term, {}, {}, {}, m, {});
end
end

Fig. 5.10. Computing Guarded Actions of the Depth (Part IV)

from the computation of the depth of S₁ is used to determine the starting conditions strt₂ := term₁ ∧ ¬strg and prmt₂ := susp ∨ abrt, which is explained as follows: If S₁ terminates and there is a strong preemption, we do not even touch S₂. However, in case of weak preemption or no preemption, strt₂ is true and triggers the execution of a surface of S₂. If such a computation is triggered, we must activate prmt₂ in case a weak preemption takes place.
sufficient to define $\text{prmt}_2 := \text{susp} \lor \text{abrt}$, since by definition of $\text{strt}_2$, we already know that the preemption that takes place (if so) must be a weak one. Finally, having computed the guarded actions of the surface of $S_2$, we compute the guarded actions of its depth. Clearly, we use the same preemption context given by the arguments $h, \text{susp}, \text{abrt}$, and $\text{strg}$ used for the call to compute the sequence $\{S_1; S_2\}$. After this, we simply collect the obtained ingredients for returning the final result.

In a similar way, the depth of a loop statement $\text{do } S \text{ while}(\sigma)$ is computed: Resuming from within the loop body $S$, we first compute its depth by a recursive call using the given arguments for $\text{susp}, \text{abrt}$, and $\text{strg}$. However, we have to increase the incarnation level to distinguish the surfaces of local declarations included in the depth of $S$. If the loop iterates, it will first compute a surface of $S$ and then again its depth. For this reason, we have to add the guards of a subsequent surface of $S$ that is executed after a previous execution of $S$ terminates. To this end, we use the start conditions $\text{strt}_1 := \text{term}_1 \land \sigma \land \neg \text{strg}$ and $\text{prmt}_1 := \text{susp} \lor \text{abrt}$ for similar reasons as explained for the sequence above. The combination of the transfer actions $G^{\text{trf}}$ and the absence actions $G^{\text{abs}}$ via function $\text{combineABS}$ is discussed in the paragraph on compilation of local declarations below.

The depth of $\text{during } S_1 \text{ do } S_2$ is determined by the depth of $S_1$ with the same arguments as given to the call, and a surface of $S_2$ that is triggered with the conditions $\text{strt} := \text{insd} \land \neg (\text{term} \lor \text{strg})$ and $\text{prmt} := \text{susp} \lor \text{abrt}$: Clearly, $S_2$ is not started if a strong preemption takes place, and otherwise only if we are inside $S_2$ and $S_1$ does not already terminate. If $S_2$ is started, we know there is no strong preemption, so that $\text{prmt} := \text{susp} \lor \text{abrt}$ denotes a weak preemption.

To explain the compilation of the preemption statements, we introduce a new statement $\text{preempt } S \text{ when}(\text{susp}, \text{abrt}, \text{strg})$ that implements one of four preemption statements according to the values of the conditions $\text{susp}, \text{abrt}, \text{strg}$. This means we have the following behavior of this statement

\[
\begin{align*}
\text{preempt } & S \\
\text{when}(\text{susp}, \text{abrt}, \text{strg}) := & \begin{cases} 
\text{S} : \text{if } \neg (\text{susp} \lor \text{abrt}) \\
\text{weak abort } S \text{ when(true)} : \text{if } \neg \text{susp} \land \text{abrt} \land \neg \text{strg} \\
\text{abort } S \text{ when(true)} : \text{if } \neg \text{susp} \land \text{abrt} \land \text{strg} \\
\text{weak suspend } S \text{ when(true)} : \text{if } \text{susp} \land \neg \text{strg} \\
\text{suspend } S \text{ when(true)} : \text{if } \text{susp} \land \text{strg}
\end{cases}
\end{align*}
\]

Note, however, that the above ‘definition’ is to be understood dynamically, i.e., if within a given environment we evaluate the conditions $\text{susp}, \text{abrt}$, and $\text{strg}$ to boolean constants, we can determine the behavior of the new statement by referring to the SOS rules of the suspension and abortion statements using the distinction with the five cases listed above. Using this statement, we can prove the following equations that explain the compilation of the preemption statements in $\text{CompDepth}$:

- $\text{preempt } \{\text{weak abort } S \text{ when}(\sigma)\} \text{ when}(\text{susp}, \text{abrt}, \text{strg}) = \text{preempt } S \text{ when}(\text{susp}, \text{abrt} \lor \sigma, \text{strg})$
• $\text{preempt } \{\text{abort } S \text{ when}(\sigma)\} \text{ when}(\text{susp, abrt, strg})$
  $= \text{preempt } S \text{ when}(\text{susp, abrt } \lor \sigma, \text{strg } \lor \sigma)$
• $\text{preempt } \{\text{weak suspend } S \text{ when}(\sigma)\} \text{ when}(\text{susp, abrt, strg})$
  $= \text{preempt } S \text{ when}(\text{susp } \lor \sigma \land \lnot \text{abrt, abrt, strg})$
• $\text{preempt } \{\text{suspend } S \text{ when}(\sigma)\} \text{ when}(\text{susp, abrt, strg})$
  $= \text{preempt } S \text{ when}(\text{susp } \lor \sigma \land \lnot \text{abrt, abrt, strg } \lor \sigma)$

For the proofs of these equations, note first that they are all true in case $\sigma$ is false, since the inner preemption statement has no effect in this case. In the remaining case, we have an environment that evaluates $\sigma = \text{true}$ so that we can replace $\sigma$ by true. Then, we have to consider five further cases that are determined by the execution context depending on the values of the preemption conditions $\text{susp, abrt}$ and $\text{strg}$. Using the SOS rules of the operational semantics, it is then finally possible to see the validity of the mentioned equations.

The above equations explain the construction of the conditions $\text{susp, abrt}$ and $\text{strg}$ used for computing guarded actions of the depth of the body of the preemption statements. The rest is trivial, except for immediate suspension statements. The depth of $\ell:\text{[weak] immediate suspend } S \text{ when}(\sigma)$ is essentially the depth of $\ell:\text{[weak] suspend } S$ which explains its computation.

It remains to explain the computation of the depth of local declarations and module calls. Clearly, the computation of the depth of a local declaration of a variable $x$ mainly consists of the depth of its body statement. The next step is to disable delayed assignments to the local variable $x$ that are executed at a point of time where the scope of $x$ is left. This has already been explained for the compilation of the surface of a local declaration. For the depth, the disabling condition $\xi := \lnot \text{susp } \land (\text{term } \lor \text{abrt})$ is used which means that there must not be an active suspension (since otherwise the scope would not be left), and either the body statement terminates or is aborted from its context. The condition $\xi$ is used to disable delayed assignments to $x$ in $D^\text{depth}$ as well as argument expressions $\tau_i$ of module calls in case that the expression $\tau_i$ is a left-hand side expression that is bound to $x$.

Besides the disabling of delayed assignments to the local variable $x$, there is another step left that is similar to the computation of the transfer action in the compilation of the surface of a local declaration: Recall that the transfer actions are of the form $(x, (\gamma, \varphi(x)))$ and have the meaning that the value of a memorized local variable $x$ right after entering its local declaration must be taken from its reincarnation $\varphi(x)$ in case $\gamma$ holds. Each surface of the local declaration of $x$ generates one transfer action, and these transfer actions are stored in the absence actions $G^\text{abs}$ that is a set of tuples $(x, cL, \text{Default}(\alpha))$ where $cL$ is a list of pairs $(\gamma_i, \varphi_i(x))$ consisting of the different transfer actions of reincarnations $\varphi_i(x)$ of $x$. Hence, function combineABS used in the compilation of the depth of loops simply adds the transfer actions $(x, (\gamma, \varphi(x)))$ on top of the corresponding list $cL$ in the absence actions. After compilation, the list
and the default value Default(α) are used to construct a case-expression that will finally be the absence action of variable x.

Finally, the depth of a module call is obtained analogously to the surface of module calls: We allocate variables \( x_{\text{insd}} \) and \( x_{\text{term}} \) to be associated with the corresponding control flow conditions by the linker when the results of the compilation of the called module is obtained. The argument expressions \( \tau_i \) are endowed by disabling conditions that are initially true. Finally, we store the start conditions \( \text{susp, abrt and strg} \) in the module call so that these can be associated with corresponding variables in the linking process.

After these explanations, we can formally state the correctness of the depth compilation in the following theorem:

**Theorem 5.2 (Compiling the Depth).** Given a Quartz statement \( S \), an incarnation index \( h \), a suspension condition \( \text{susp} \), an abortion condition \( \text{abrt} \), a strength condition \( \text{strg} \), the call \( \text{CompDepth}(h, \text{susp}, \text{abrt}, \text{strg}, S) \) computes a tuple \((L, \text{insd}, \text{term}, D_{\text{depth}}, G_{\text{abs}}, M_{\text{src}}, M_{\text{depth}}, C_{\text{depth}})\) with the following meaning (taking the associations of variables into account):

- \( L \) is the set of local declarations inside \( S \)
- \( \text{insd} \) is equivalent to \( \text{insd}(S) \)
- \( \text{term} \) is equivalent to \( \text{term}(S) \)
- \( D_{\text{depth}} = \text{ActDepth}(S) \), i.e., the set of guarded actions of the data flow in the depth of \( S \)
- \( G_{\text{abs}} \) is the set of absence reactions for the locally declared variables which are tuples \((x, cL, \text{Default}(\alpha))\) where \( cL \) is a list of pairs \((\gamma_i, \varrho_i(x))\) that holds the information to generate the absence reaction of the local variable \( x \),
- \( M_{\text{src}} \) is the set of module calls to surfaces of other modules inside \( S \) whose content is as described in Theorem 5.1,
- \( M_{\text{depth}} \) is the set of module calls to depths of other modules inside \( S \) with the following components:
  - \( \text{nameI} \) is the name of the instance of the module call,
  - \( \text{nameM} \) is the name of the called module,
  - \( x_{\text{insd}} \) is the variable that is used as placeholder of the inside condition \( \text{insd} \) of the module and will be associated with that condition once it is known,
  - \( x_{\text{term}} \) is the variable that is used as placeholder of the termination condition \( \text{term} \) of the module call,
  - \( \text{susp} \), \( \text{abrt} \) and \( \text{strg} \) are the preemption conditions of the module call,
  - and \( \text{argL} \) is a list of pairs \((\tau_i, \varphi_i)\) where \((\tau_1, \ldots, \tau_n)\) are the renamed expressions used in the module call and \( \varphi_i \) are additional disabling conditions to be added to the guards of variables that are locally declared.
- \( C_{\text{depth}} \) are the guarded actions of the control flow which is a possibly empty set of guarded actions of the form \((\gamma, \text{next}(\ell) = \text{true})\). If \( C_{\text{depth}} = \{\} \), then \( S \) has no locations, and \( \text{move}(S) \) is false. Otherwise, we have the following equality:
\[
\text{move } (S') \leftrightarrow \bigwedge_{(\gamma, \text{next}(\ell) = \text{true}) \in C^{\text{dpth}}} \gamma \rightarrow \text{next}(\ell)
\]

5.3 Overall Compilation and Linking

In the previous section, we have discussed the functions \text{CompSurface} and \text{CompDepth} to compute the guarded actions of the surface and the depth of a statement. These functions have to be called by an overall function \text{Compile} that has to provide the additional arguments for the calls to \text{CompSurface} and \text{CompDepth}. In particular, this function has to provide an associative store that is used in the functions \text{CompSurface} and \text{CompDepth}.

For dealing with associative stores, we assume the following functions that can be easily implemented via hash tables or binary search trees:

- Function \text{NewAssociativeStore} generates a new associative store to which the following calls to \text{NewDef} or \text{NewVar} refer to.
- Function \text{NewVar}() generates a new variable in the current associative store without already associating a boolean expression with it. Such an association can be made later by function call \text{NewAssoc}(x, \varphi).
- Function \text{NewDef}(\varphi) checks whether \varphi is already associated with a variable in the current associative store. If this is the case, this variable is returned, otherwise, a new variable \( x \) is generated by \text{NewVar}() and associated with \varphi by function \text{NewAssoc}(x, \varphi). Finally, the generated variable \( x \) is returned.
- Function \text{GetExpr}(x) returns the expression associated with variable \( x \).

Figure 5.11 shows the overall compile function. First, it allocates a new associative store by calling function \text{NewAssociativeStore}. After this, new variables \( x_{\text{strt}} \) and \( x_{\text{prmt}} \) are allocated for the context conditions \( \text{strt} \) and \( \text{prmt} \) of the call to function \text{CompSurface}. Using the empty renaming, the call to \text{CompSurface} then computes the guarded actions of the surface \( T_{\text{surf}} \).

After the surface has been compiled, the analogous steps are performed for the compilation of the depth: First, new variables \( x_{\text{susp}}, x_{\text{abrt}} \) and \( x_{\text{strg}} \) are allocated that are used as the compilation context \( \text{susp}, \text{abrt} \) and \( \text{strg} \) of the function call to \text{CompDepth}. Using the incarnation level 0, the call to \text{CompDepth} computes then the guarded actions of the depth \( T_{\text{dpth}} \).

Finally, all results are collected in a tuple that is returned. It has to be emphasized that the ordering of the actions related with the associative store has been chosen with some care: Clearly, it would be possible to first allocate all the variables \( x_{\text{strt}}, x_{\text{prmt}}, x_{\text{susp}}, x_{\text{abrt}} \) and \( x_{\text{strg}} \) in a single block before calling functions \text{CompSurface} and \text{CompDepth}. However, there is a good reason for using the code as given in Figure 5.11: Typically, the variables generated by the associative store are generated along a total ordering. For example, in the current implementation of the Averest system, the generated variables are obtained by applying a constructor function to an integer index so that the
5.3 Overall Compilation and Linking

function Compile(S)
    NewAssociativeStore();
    // compiling the surface
    x_{strt} = NewVar();
    x_{prmt} = NewVar();
    T_{surf} := CompSurface({}, x_{strt}, x_{prmt}, S);
    // compiling the depth
    x_{susp} = NewVar();
    x_{abrt} = NewVar();
    x_{strg} = NewVar();
    T_{depth} := CompDepth(0, x_{susp}, x_{abrt}, x_{strg}, S);
    // preparing return value
    A := ContentsOfAssociativeStore();
    C := (x_{strt}, x_{prmt}, x_{susp}, x_{abrt}, x_{strg});
    return (A, C, T_{surf}, T_{depth});
end

Fig. 5.11. Overall Compile Function

generated variables are essentially integers. The function NewVar then simply
increments the current index, and similarly, function NewDef may simply not
modify the store or it will increment the index and add a new association for
the generated variable.

Now, if the code is implemented as given in Figure 5.11, one can detect
‘breakpoints’ in the associative store, where a breakpoint is determined by a
variable that is not yet associated with a boolean condition. These breakpoints
results only from calls to function NewVar, and inspecting the code reveals
that such calls occur only in the following functions:

- in function ModuleCallCompSurface to allocate $x_{inst}$
- in function ModuleCallCompDepth to allocate $x_{insd}$ and $x_{term}$
- in function Compile before the call to CompSurface to allocate $x_{strt}$ and
  $x_{prmt}$, and
- in function Compile before the call to CompDepth to allocate $x_{susp}$, $x_{abrt}$
  and $x_{strg}$.

Hence, the detection of breakpoints allows one to find the places in the as-
sociative store, where a module call or a call to compile the overall surface
or depth of the given module are made. Clearly, the first two allocated vari-
ables in the associative store are $x_{strt}$ and $x_{prmt}$, so that these breakpoints
are already the first two variables in the store. Moreover, we know which of
the associations were only made for the surface since the variable $x_{susp}$ is the
first one allocated by the compilation of the depth. These observations allows
one to implement the expansion of surface module calls more efficiently since
it is then sufficient to only integrate the surface part of the corresponding
associated store.
It is also important to detect the remaining breakpoints that are due to module calls, since the associative stores of the called modules can then be integrated at the places where the module call has been reached by the compile functions. The result will then be exactly the same as if the compiler would have expanded the module at compile time, and therefore the associative store keeps its associations with a read-after-write ordering: All variables that occur in an expression $\varphi$ that is associated with some variable $x_\varphi$ are less than the variable $x_\varphi$ and thus, their associations occur before the association of $x_\varphi$. This is a very important feature for code generation, since the code can be evaluated in this ordering. Of course, one can alternatively apply a topological sort using the dependency ordering, but this is not necessary with the tricks mentioned above. We will discuss the use of the breakpoints in the linker below, but first leave out this issue.

In the Averest system, the tuple $(A, T_{srfc}, T_{dpth})$ generated by the compiler is stored in a file that can be read later by the linker. If the linker is called for such a module, it will access the corresponding file to read the tuple $(A, T_{srfc}, T_{dpth})$ generated previously by the compiler. When the linker has read the tuple, it can determine the final compile context. Recall that the compiler has only allocated variables $x_{strt}$, $x_{prmt}$, $x_{susp}$, $x_{abrt}$, and $x_{strg}$, but did not already associated them with boolean expressions. The compiler cannot make this association since it does not know the context in which the module is called, and indeed, there could be many contexts since there can be many calls to the same module. However, if the linker is called for a module, this context of the outermost module can be determined as follows:

- $x_{strt} = true$
- $x_{prmt} = false$
- $x_{susp} = false$
- $x_{abrt} = false$
- $x_{strg} = false$

However, simply associating these variables with boolean constants would leave out important optimizations: The linker should propagate these constants through the associative store, since many of the expressions stored there will thereby be reduced to boolean constants, so that their association with variables become unnecessary.

During the propagation of boolean constants through the associative store, the linker has to check for a potential breakpoint. If a breakpoint is reached that belongs to a module call, the linker has to recursively link the code of the called module.

We therefore make use of the function Propagate shown in Figure 5.15 to generate a new store and a substitution $\varphi$. The substitution $\varphi$ maps variables of the old associative store either to variables of the new store or to one of the boolean constants true and false. Hence, $\varphi$ is a translation from the old to the new associative store.
function Link(nameM)
  // read data of compiled module
  (A, C, T^{src}, T^{dpth}) := ReadAIF(nameM);
  (x_{strt}, x_{prmt}, x_{susp}, x_{abrt}, x_{strg}) := C;
  (inst, D^{src}_{T}, G^{trf}_{T}, M^{src}_{T}, C^{src}_{T}) := T^{src};
  (L, insd, term, D^{dpth}_{L}, G^{abs}_{L}, M^{dpth}_{L}, C^{dpth}_{L}) := T^{dpth};
  M^{src} := M^{src}_{a} \cup M^{src}_{b};
  // determine given compilation context
  C^{\varphi} := \{\};
  NewAssoc(x_{strt}, true); \varphi := \varphi \cup \{(x_{strt}, true)\};
  NewAssoc(x_{prmt}, false); \varphi := \varphi \cup \{(x_{prmt}, false)\};
  NewAssoc(x_{susp}, false); \varphi := \varphi \cup \{(x_{susp}, false)\};
  NewAssoc(x_{abrt}, false); \varphi := \varphi \cup \{(x_{abrt}, false)\};
  NewAssoc(x_{strg}, false); \varphi := \varphi \cup \{(x_{strg}, false)\};
  // remaining breakpoints are due to module calls
  NewAssociativeStore();
  L^{src} := (\{\}, \{\}, \{\});
  L^{dpth} := (\{\}, \{\}, \{\}, \{\}, \{\});
  (\varphi, L^{src}, L^{dpth}) := Propagate(\varphi, A, M^{src}, M^{dpth});
  (D^{src}_{T}, G^{C^{src}_{T}}_{T}, C^{src}_{T}) := L^{src};
  (D^{dpth}_{L}, G^{abs}_{L}, C^{dpth}_{L}) := L^{dpth};
  // substitute guarded actions of this module and add the imported actions
  D^{src} := \varphi(D^{src}_{T}) \cup D^{src}_{L};
  G^{trf} := \varphi(G^{trf}_{T}) \cup G^{trf}_{L};
  C^{src} := \varphi(C^{src}_{T}) \cup C^{src}_{L};
  D^{dpth} := \varphi(D^{dpth}_{L}) \cup D^{dpth}_{L};
  G^{abs} := \varphi(G^{abs}_{L}) \cup G^{abs}_{L};
  C^{dpth} := \varphi(C^{dpth}_{L}) \cup C^{dpth}_{L};
  // determine control flow constants
  NewAssoc(instant, inst);
  NewAssoc(inside, insd);
  NewAssoc(terminate, term);
  // collect results
  A := ContentsOfAssociativeStore();
  T^{src} := (D^{src}, C^{src});
  T^{dpth} := (D^{dpth}, C^{dpth});
  G^{abs} := combineABS(G^{trf}, G^{abs});
  return (L, A, T^{src}, T^{dpth}, G^{abs});
end

Fig. 5.12. Propagating the Compile Context through the Associative Store
function Propagate(\( \rho, L_{src}, L_{depth}, A, M_{src}, M_{depth} \))
if(null(\( A \)))
    // linking complete; the associative store has been elaborated
    return (\( \rho, L_{src}, L_{depth} \));
else
    \( (x, \varphi) := \text{head}(A) \);
    \( A := \text{tail}(A) \);
    if(isBreakpoint(\( x, \varphi \)))
        // find corresponding module call
        \( M_{src} := \{ (m_{name}, m_{nameM}, x_{inst}, m_{str}, m_{prmt}, m_{argL}) \in M_{src} \mid x_{inst} = x, \varphi \} \);
        \( M_{depth} := \{ (m_{name}, m_{nameM}, x_{insd}, x_{term}, m_{sus}, m_{abrt}, m_{str}, m_{argL}) \in M_{depth} \mid x_{insd} = x, \varphi \} \);
        // here, either \( M_{src} = \{ m \} \land M_{depth} = \{ \} \) or \( M_{src} = \{ \} \land M_{depth} = \{ m \} \) holds
        if(null(\( M_{depth} \)))
            // link surface module call
            \((D_{src}^{L}, G_{src}^{L}, C_{src}^{L}) := L_{src},\)
            \(\{ (m_{name}, m_{nameM}, x_{inst}, m_{str}, m_{prmt}, m_{argL}) \} := M_{src},\)
            \((\text{insd}, D_{src}^{L}, G_{src}^{L}, C_{src}^{L}) := \text{LinkSurfaceModule}(m_{str}, m_{prmt}, m_{nameM})\);
            \(\text{NewAssoc}(x_{inst}, \{ \text{insd} \});\)
            \(\varphi := \varphi \cup \{ \{ x_{inst}, \text{insd} \} \};\)
            \(L_{src} := (D_{src}^{L} \cup D_{src}^{L}, G_{src}^{L} \cup G_{src}^{L}, C_{src}^{L} \cup C_{src}^{L});\)
            return Propagate(\( \rho, L_{src}, L_{depth}, A, M_{src} \setminus M_{src}, M_{depth} \))
else
    // link depth module call
    \((D_{depth}^{L}, G_{depth}^{L}, C_{depth}^{L}) := L_{depth},\)
    \(\{ (m_{name}, m_{nameM}, x_{insd}, x_{term}, m_{sus}, m_{abrt}, m_{str}, m_{argL}) \} := M_{depth},\)
    \((\text{term}, D_{depth}^{L}, G_{depth}^{L}, C_{depth}^{L}) := \text{LinkDepthModule}(m_{sus}, m_{abrt}, m_{str}, m_{nameM})\);
    \(\text{NewAssoc}(x_{insd}, \{ \text{term} \});\)
    \(\text{NewAssoc}(x_{term}, \{ \text{term} \});\)
    \(\varphi := \varphi \cup \{ \{ x_{insd}, \text{insd} \}, \{ x_{term}, \text{term} \} \};\)
    \(L_{depth} := (D_{depth}^{L} \cup D_{depth}^{L}, G_{depth}^{L} \cup G_{depth}^{L}, C_{depth}^{L} \cup C_{depth}^{L});\)
    return Propagate(\( \rho, L_{src}, L_{depth}, A, M_{src} \setminus M_{src}, M_{depth} \))
else
    // no module call, thus process association (x, \varphi)
    \( \psi := \text{BoolSimp}(\rho(\varphi)) \);
    case \( \psi \) of
    true : \( \varphi := \varphi \cup \{ x_{true}, \text{true} \}; \)
    false : \( \varphi := \varphi \cup \{ x_{false}, \text{false} \}; \)
    isVar(\( \psi_x \)) : \( \psi := \varphi \cup \{ x_{\psi_x}, \psi \}; \)
else
    \( y_{\psi} := \text{NewDef}(\psi) \);
    \( \varphi := \varphi \cup \{ x_{\psi}, y_{\psi} \}; \)
end
end
return Propagate(\( \rho, L_{src}, L_{depth}, A, M_{src}, M_{depth} \));
end

Fig. 5.13. Propagating the Compile Context through the Associative Store
5.3 Overall Compilation and Linking 175

Function BoolSimp applies thereby the following boolean simplifications to the topmost boolean operator of the expression it is applied to:

- ¬true = false
- ¬false = true
- ¬¬ϕ = ϕ
- true ∧ ψ = ψ
- false ∧ ψ = false
- ϕ ∧ true = ϕ
- ϕ ∧ false = false
- ϕ ∧ ¬ϕ = false
- ¬ϕ ∧ ψ = false
- true ∨ ψ = true
- false ∨ ψ = ψ
- ϕ ∨ true = true
- ϕ ∨ false = ϕ
- ϕ ∨ ¬ϕ = true
- ¬ϕ ∨ ϕ = true

Having computed a new associative store, and the substitution ϱ, the other results of the compiler are substituted.

1. substitute everything with ϱ
2. combine Gtrf and Gabs with CombineABS
3. read in tuple of called modules and integrate their associative stores

Having the new store and the substitution ϱ, the linker will next use the substitution to substitute the guarded actions. Clearly, guarded actions whose guards are reduced to false, are left out.

Since the some of the associated variables are no longer used

Finally, reconsider the compilation of sequences \{S_1; S_2\}. The definition of strt_2 and prmt_2 in the depth compilation of a sequence \{S_1; S_2\}: We use \texttt{strt}_2 := \texttt{term}_1 ∧ ¬\texttt{strg} and \texttt{prmt}_2 := \texttt{susp} ∨ \texttt{abrt}. The surface of the sequence \{S_1; S_2\} will also compute a surface of S_2, and the question to be discussed here is whether these two surfaces can be both executed within a macro step, and if so, if reincarnation of local variables needs to be handled here.

Indeed, both surfaces can be executed within a macro step: Figure 5.16 shows a module that demonstrates this behavior: If the control is currently at location w1 and x holds, we execute \texttt{b = true} but do not rest at w2 (due to the weak abortion). Hence, the loop restarts its body, but this time the abortion statement does not react (since it is not an immediate abortion). Hence, we execute \texttt{a = true}, and since x holds, we do not rest at w1. Consequently, we execute \texttt{b = true} twice and reach then location w2:pause.
function LinkSurfaceModule(strt, prmt, nameM)
// read data of compiled module
(A, C, Tsrc, Tdpth) := ReadAIF(nameM);
(xstrt, xprmt, xsusp, xabrt, xstrg) := C;
(inst, Dsrc, Gtrf, Msrc, Csrc) := Tsrc;
(L, insd, term, Ddpth, Cabs, Msrfc, Msrc, Csrc, Tdpth) := Tdpth;
Msrc := Msrfc \cup Msrc;
// determine given compilation context
\( \varrho := \{(x_{\text{strt}}, \text{strt}), (x_{\text{prmt}}, \text{prmt})\} \);
NewAssoc(xstrt, strt);
NewAssoc(xprmt, prmt);
NewAssoc(xsus, false);
NewAssoc(xabrt, false);
NewAssoc(xstrg, false);
// remaining breakpoints are due to module calls
Lsrc := (\{\}, \{\}, \{\});
Ldpth := (\{\}, \{\}, \{\});
(\varrho, Lsrc, Ldpth) := Propagate(\varrho, Lsrc, Ldpth, A, Msrc, Mdpth);
(Dsrc, Gtrf, Csrc) := Lsrc;
// substitute guarded actions of this module and add the imported actions
Dsrc := \varrho(Dsrc) \cup Dsrc;
Gtrf := \varrho(Gtrf) \cup Gtrf;
Csrc := \varrho(Csrc) \cup Csrc;
// return results
return (\varrho(inst), Dsrc, Gtrf, Csrc)
end

Fig. 5.14. Propagating the Compile Context through the Associative Store
function LinkDepthModule(susp, prmt, strg, nameM)
   // read data of compiled module
   (A, C, Tsrc, Tdepth) := ReadAIF(nameM);
   (xstrt, xprmt, x susp, x abrt, xstrg) := C;
   (inst, Tsrc, Tdpth, Msrfc, Cdepth, Msrfc) := Tsrc;
   (C, insd, term, Tdpth, Gabs, Msrcf, Msrfc, Tdepth) := Tdepth;
   Msrfc := Msrfc \cup Msrcf;
   // determine given compilation context
   \( \varrho \) := \{(x susp, susp), (x abrt, abrt), (xstrg, strg)\};
   NewAssoc(x strt, false);
   NewAssoc(x prmt, false);
   NewAssoc(x susp, false);
   NewAssoc(x abrt, false);
   NewAssoc(x strg, false);
   // remaining breakpoints are due to module calls
   Lsrc := (\{\}, \{\}, \{\});
   Ldpth := (\{\}, \{\}, \{\});
   (\varrho, Lsrc, Ldpth) := Propagate(\varrho, Lsrc, Ldpth, A, Msrfc, Msrfc);
   \( \varrho, Ldpth \) := Propagate(\varrho, Ldpth, A, Msrfc, Msrfc);
   // substitute guarded actions of this module and add the imported actions
   Ddpth := \varrho(Ddpth) \cup Tdpth;
   Gabs := \varrho(Gabs) \cup Gabs;
   Cdepth := \varrho(Cdepth) \cup Cdepth;
   // return results
   return (\varrho(insd), \varrho(term), Ddpth, Gabs, Cdepth)
end

Fig. 5.15. Propagating the Compile Context through the Associative Store

module M(event bool ?x,!a,!b) {
   do
      weak abort
      S1 {a = true;
           if(!x) w1:pause;
           S2 {b = true;
                w2:pause;
                when(x)
                while(true)
           }
}

Fig. 5.16. A module that starts the surface of \( S_2 \) twice in a macro step.
6

Synthesis of Synchronous Systems

6.1 Synthesis of Data Flow Equations

Using the algorithms of the previous section, we can compute the control and data flow of every Quartz program. In particular, the function StartCompile as given in Figure ?? computes a tuple \((\Xi, G, R)\) where \(G\) and \(R\) represent the data and the control flow, respectively. The control flow \(R\) is thereby already a set of transition equations that contains for every location \(\ell\) of the program a transition equation of the form \(\text{next}(\ell) = \varphi\). These transition equations can be directly used for code generation as their right hand sides do not refer to future values. Hence, the right hand sides can be evaluated in the current variable assignment to determine the locations that hold the control flow at the next point of time.

However, the data flow is still represented by the guarded actions \(G\) that are also computed by the function StartCompile. In contrast to the control flow, the data flow is therefore not yet in an executable form: reaction-to-absence, potential write conflicts and causality cycles are further problems for code generation. For this reason, additional checks are necessary to complete the code generation. In the following, we describe how initial and transition equations can be derived from the guarded commands, provided that the program has no write conflicts. Hence, the data flow is then also given in terms of an equation system. Checking write conflicts is considered in Chapter 8, and the symbolic causality analysis is be considered in Section ??.

Assume we have computed for a local variable \(x\) and its \(d\) reincarnations \(x_1, \ldots, x_d\) the following guarded actions, where \(x_d\) is assumed to be the outermost reincarnation of \(x\):

\[
\begin{align*}
(\chi_1, \text{next}(x) = \pi_1), & \ldots, (\chi_q, \text{next}(x) = \pi_q) \\
(\gamma_1, x = \tau_1), & \ldots, (\gamma_p, x = \tau_p), \\
(\gamma_{1,1}, x_1 = \tau_{1,1}), & \ldots, (\gamma_{p,1}, x_1 = \tau_{p,1}) \\
& \ldots \\
(\gamma_{d,1}, x_d = \tau_{d,1}), & \ldots, (\gamma_{d,p_d}, x_d = \tau_{d,p_d})
\end{align*}
\]
Note that reincarnations $x_i$ do always have event storage mode, since they only exist for one point of time. Note further that we have no delayed actions on reincarnated variables, since the renamings due to the incarnation level function $h$ have not been applied on the left hand sides of delayed assignments.

For this reason, the equation of a reincarnated variable $x_i$ is defined as the invariant $\text{Invar}_{x_i}$ in Figure 6.1 that has to hold at every point of time. This invariant simply states that whenever the trigger condition $\gamma_{i,j}$ holds, then the corresponding equation $x_i = \tau_{i,j}$ must also hold. If no trigger condition $\gamma_{i,j}$ holds, then it is required that $x_i$ equals to the default value $\text{Default}(x_i)$. The ordering of the assignments in the case construct is arbitrary. Since we assume that there are no write conflicts, at most one of the guards $\gamma_{i,j}$ should be active.

The transition equation for $x$ is more difficult, since a straightforward construction would refer to values of the trigger conditions of immediate assignments at the next point of time. To circumvent this, we introduce an auxiliary variable $x'$, called the carrier of $x$ to capture delayed assignments at the previous point of time. Before considering the equations for $x'$, let us consider the invariant $\text{Invar}_x$ of $x$ in terms of $x'$ as given in Figure 6.1. Clearly, we have to demand that $x$ equals to $\tau_i$ whenever the guard $\gamma_i$ of an immediate assignment $x = \tau_i$ holds. In case no guard of an immediate assignment to $x$ holds, we have to distinguish between the cases where $x$ is an event or a state variable. These cases are considered by different equations of the carrier variable $x'$, so that we can simply take the value of $x'$ in this case.

Hence, it remains to define equations for the carrier variable $x'$ depending on whether $x$ is an event or a memorized variable. The meaning of $x'$ is as follows: $x'$ captures all of the delayed assignments $\text{next}(x) = \pi_j$ to $x$, that is whenever $\text{next}(x) = \pi_j$ is executed, we evaluate the right hand side $\pi_j$ at the current point of time and assign this value to $x'$ (not yet to $x$) at the next point of time. Hence, $x'$ is determined by the delayed assignments to $x$. This leaves open what the initial value of $x'$ should be, so we additionally define the initial value of $x'$ is the default value of $x$.

By this definition of the initial value of $x'$, also the initial value of $x$ is correct. Next, consider a point of time $t > 0$. Then, if one of the immediate assignments to $x$ is enabled, then this assignment determines the value of $x$ at this point of time as given by the invariant equation $\text{Invar}_x$. Otherwise, a delayed assignment $\text{next}(x) = \pi_j$ may have been executed at the previous point of time. If so, then $x'$ has now the value that has been obtained by evaluating $\pi_j$ at the previous point of time, and the invariant equation $\text{Invar}_x$ takes this value via $x'$.

Finally, if neither an immediate assignment $x = \tau_i$ is currently executed nor a delayed assignment $\text{next}(x) = \pi_j$ has been been executed at the previous point of time, we have to refer to the reaction-to-absence, which is different for event and memorized variables. For an event variable, we simple use the default value of $x$ that is also transfered via $x'$ to $x$. 
Computed Guarded Commands of $x$ and its Reincarnations $x_1, \ldots, x_d$:

\[(\chi_1, \text{next}(x) = \pi_1), \ldots, (\chi_q, \text{next}(x) = \pi_q)\]
\[(\gamma_{1,1}, x_1 = \tau_{1,1}), \ldots, (\gamma_{p,1}, x_1 = \tau_{1,p})\]
\[\vdots \quad \vdots\]
\[(\gamma_{d,1}, x_d = \tau_{d,1}), \ldots, (\gamma_{d,p}, x_d = \tau_{d,p})\]

Invariant Equation $\text{Invar}_{x_i}$ of Reincarnated Variable $x_i$:

\[
x_i = \begin{cases} 
\text{case} \\
\gamma_{i,1} : \tau_{i,1}; \\
\vdots \\
\gamma_{i,p} : \tau_{i,p}; \\
\text{else Default}(x) 
\end{cases} 
\]

Invariant Equation $\text{Invar}_x$ of Variable $x$:

\[
x = \begin{cases} 
\text{case} \\
\gamma_1 : \tau_1; \\
\vdots \\
\gamma_p : \tau_p; \\
\text{else } x' 
\end{cases} 
\]

Initial Equation $\text{Init}_{x'}$, of Carrier $x'$:

\[x' = \text{Default}(x)\]

Transition Equation $\text{EventTrans}_{x'}$, of Carrier $x'$ for Event Variable $x$:

\[\text{next}(x') = \begin{cases} 
\text{case} \\
\chi_1 : \pi_1; \\
\vdots \\
\chi_q : \pi_q; \\
\text{else Default}(x) 
\end{cases} 
\]

Transition Equation $\text{MemTrans}_{x'}$, of Carrier $x'$ for Memorized Variable $x$:

\[\text{next}(x') = \begin{cases} 
\text{case} \\
\chi_1 : \pi_1; \\
\vdots \\
\chi_q : \pi_q; \\
go_d : x_d; \\
\vdots \\
go_1 : x_1; \\
\text{else } x 
\end{cases} 
\]

Fig. 6.1. Equations to Define the Data Flow of Local Event/Memorized Variables
According to the semantics, a memorized output variable $x$ (that has no reincarnations), should maintain its value if the reaction to absence has to be applied. The equations of Figure 6.1 correctly implement this behavior, since $x'$ has stored the previous value of $x$, which is therefore forwarded to $x$ by $\text{Invar}_x$.

Finally, if $x$ is a memorized local variable, then the reaction-to-absence is more difficult: We have to check whether the scope of the local variable has been re-entered at the previous point of time. In this case, we have to consult the value of the most recently re-entered incarnation of the local declaration of the previous point of time to determine the current value of $x$. Fortunately, we have stored in $x'$ this value in case that no delayed action has also been executed at the previous point of time. Hence, we can transfer this value via $x'$ to $x$.

**Computed Guarded Commands of $x$ and its Reincarnations $x_1, \ldots, x_d$:**

$$(\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p),$$

$$(\gamma_1, x_1 = \tau_{1,1}), \ldots, (\gamma_1, p_1, x_1 = \tau_{1,p_1})$$

$$\vdots$$

$$(\gamma_d, x_d = \tau_{d,1}), \ldots, (\gamma_d, p_d, x_d = \tau_{d,p_d})$$

**Invariant Equation $\text{Invar}_{x_i}$ of Reincarnated Variable $x_i$:**

$$x_i = \begin{cases} 
\text{case} \\
\gamma_{i,1} : \tau_{i,1} \\
\vdots \\
\gamma_{i,p_i} : \tau_{i,p_i} \\
\text{else Default}(x) 
\end{cases}$$

**Invariant Equation $\text{Invar}_x$ of Event Variable $x$:**

$$x = \begin{cases} 
\text{case} \\
\gamma_1 : \tau_1 \\
\vdots \\
\gamma_p : \tau_p \\
\text{else Default}(x) 
\end{cases}$$

Fig. 6.2. Equations to Define the Data Flow of Local Event Variables without Delayed Assignments

Finally, let us consider two important special cases: First, Figure 6.2 considers the case, where no delayed assignments occur for an event variable. In this case, it is possible to only generate invariant equations for $x$ and its reincarnated variables $x_i$ without using a carrier variable $x'$. The important fact is here that no storage is required, i.e., for a hardware circuit synthesis,
6.1 Synthesis of Data Flow Equations

we need no registers, and for a transition system, there is no need to generate states.

**Computed Guarded Commands of** \( x \) **and its Reincarnations** \( x_1, \ldots, x_d \):

\[ (\chi_1, \text{next}(x) = \pi_1), \ldots, (\chi_q, \text{next}(x) = \pi_q) \]

**Initial Equation** \( \text{Init}_x \) **for Variable** \( x \):

\[ x = \text{Default}(x) \]

**Transition Equation** \( \text{EventTrans}_x \) **for Event Variable** \( x \):

\[
\text{next}(x) = \begin{cases} 
\chi_1 : \pi_1; \\
\vdots \\
\chi_q : \pi_q; \\
\text{else} \ \text{Default}(x)
\end{cases}
\]

**Transition Equation** \( \text{MemTrans}_{x'} \) **for Memorized Variable** \( x \):

\[
\text{next}(x) = \begin{cases} 
\chi_1 : \pi_1; \\
\vdots \\
\chi_q : \pi_q; \\
g_{o_d} : \text{Default}(x); \\
\vdots \\
g_{o_1} : \text{Default}(x); \\
\text{else} \ x
\end{cases}
\]

**Fig. 6.3.** Equations to Define the Data Flow of Local Variables without Immediate Assignments

The other special is considered in Figure 6.3, where it is assumed that no immediate assignments appear in the program. In this case, there are no assignments on the reincarnations \( x_i \), so that these do all have the default value. As a consequence, the invariant equation that defines \( x \) is simply \( x = x' \), so that we conclude that we need no carrier variable \( x' \), since \( x \) itself has that behavior. As a consequence, we obtain the simplified equations as given in Figure 6.3.

To compute the runtimes of the algorithms of the previous section, let \( T_{\text{Eqs}}(n) \) be the maximal time necessary to compile programs \( S \) of length \( |S| \leq n \). It is clear that due to the recursive implementation, \( T_{\text{Eqs}}(n + 1) \) will first require time \( T_{\text{Eqs}}(n) \) plus some time for additional computations. For most statements this additional time is just a constant. However, sequences, loops, immediate suspension, and the during statement require an additional call to the surface compilation. We already know that this requires – due to sharing
common subexpressions of the control flow predicates – time $O(|S|)$. For this reason, the compilation of the depth of a statement $S$ requires time $O(|S|^2)$.

Finally, we have to consider assignments where the left hand sides are not simply variables. The left hand side might also be an array access, a tuple access, a bitvector slice, or even a combination of these expressions. In these cases, the construction of the equations is done as shown above. We simply have to treat the left hand sides are atomic entities, where we have the following restrictions:

- For a tuple access, we can simply treat the expression $x.1$ as a variable of atomic type, for which we can use the equation synthesis as described in this section.
- For bitvector slices, the index expressions are static expressions, so that we can generate for every selected bit the corresponding boolean expression of the right hand side. We can then simply assign values for the different bits, but note that bitvectors are atomic types, and therefore the bitvector variable $x$ is assigned a value instead of its single bits.
- Arrays impose a major problem for the equational synthesis, since the left hand side may use a dynamic expression $\pi$ as index expression whose value is not known at compile time. For this reason, we first have to apply the following operation to all guarded actions of array variables: Consider a one-dimensional array $x$ with $\delta$ entries. Then, we consider the variables $x_0, \ldots, x_{\delta-1}$ instead of $x$, and replace the guarded action $(\gamma, x[\pi]=\tau)$ by $(\gamma \land (\pi = 0), x_0 = \tau), \ldots, (\gamma \land (\pi = \delta - 1), x_{\delta-1} = \tau)$, and analogously for delayed assignments. Hence, every assignment to an array variable is multiplied for all array entries, which make the equational synthesis bad for arrays (even though the expression $\pi$ can be shared and must therefore be only once evaluated)!

6.1.1 Equation Systems as Hardware Circuits

In the previous subsections, we have explained how the control and data flow a Quartz program can be translated to a single equation system. The equation system for the control flow thereby consists of exactly one transition equation of the form $\text{next}(\ell) = \varphi$ for every program location $\ell$. The data flow may introduce additional carrier variables $x'$ as storage for memorized variables and event variables with delayed assignments. Then, we have for every carrier variable an initial equation and a transition equation, and for the original output variable, we have an additional invariant equation.

In the following, we use a vector notation $\overrightarrow{x}$ to abbreviate sets of variables like $x_0, \ldots, x_n$. If we consider a Quartz program with locations $\overrightarrow{\ell}$, input variables $\overrightarrow{x}$, output/local variables $\overrightarrow{y}$, reincarnated variables $\overrightarrow{y'}$, and introduced carrier variables $\overrightarrow{z}$, then we obtain in the general case (we ignore the optimized equation systems given in Figure 6.2 and Figure 6.3) invariant, initial, and transition equations of the following forms:
6.1 Synthesis of Data Flow Equations

\[
\begin{align*}
\text{init}(\vec{\ell}) &= \vec{a} \\
\text{init}(\vec{\omega}) &= \vec{b} \\
\text{next}(\vec{\ell}) &= f(\vec{\omega}, \vec{\ell}, \vec{y}, \vec{y}') \\
\text{next}(\vec{\omega}) &= g(\vec{\omega}, \vec{\ell}, \vec{y}, \vec{y}') \\
\text{invar}(\vec{y}) &= h(\vec{\omega}, \vec{\ell}, \vec{y}, \vec{y}', \vec{\omega}) \\
\text{invar}(\vec{y}') &= h'(\vec{\omega}, \vec{\ell}, \vec{y}, \vec{y}')
\end{align*}
\]

The vectors \(\vec{a}\) and \(\vec{b}\) are thereby constant. The location variables \(\vec{\ell}\) and the carrier variables \(\vec{\omega}\) are thereby the state variables of a state machine, hence, these variables build the state space of the program. In contrast, the local, reincarnated, and output variables are defined in terms of the location, carrier, and input variables.

By construction, neither the initial nor the transition equations suffer from cyclic dependencies. However, combinational cycles may occur in the invariant equations, which leads to the definition of cyclic and acyclic programs. To this end, there is no benefit in distinguishing between location and carrier variables, and also the distinction between output, local, and reincarnated variables is not valuable. For the causality analysis that is necessary to remove these cycles, we can therefore abstract to simpler equation systems of the following form that describes a Mealy machine if there were no cycles in the invariant part:

\[
\begin{align*}
\text{init}(\vec{\omega}) &= \vec{y}' \\
\text{next}(\vec{\omega}) &= \Delta(\vec{\omega}, \vec{\omega}, \vec{y}) \\
\text{invar}(\vec{y}) &= \Gamma(\vec{\omega}, \vec{\omega}, \vec{y})
\end{align*}
\]

The symbolic causality analysis that we describe in Section ?? is able to remove cycles that may occur in these equation systems, so that for constructive programs, we can even assume that we obtain equation systems of the form

\[
\begin{align*}
\text{init}(\vec{\omega}) &= \vec{y}' \\
\text{next}(\vec{\omega}) &= \Delta(\vec{\omega}, \vec{\omega}, \vec{y}) \\
\text{invar}(\vec{y}) &= \Gamma(\vec{\omega}, \vec{\omega})
\end{align*}
\]

It is obvious that these equation systems are already descriptions of corresponding hardware circuits, provided that the operations that occur in the equations can be mapped to hardware: The state variables \(\vec{\omega}\) are outputs of registers, while the other variables \(\vec{y}\) are outputs of combinational gates. The compilation into equation systems is therefore essentially also a compilation into hardware circuits.

It is interesting to reconsider the compilation of the surface and depth parts in terms of hardware synthesis. To this end, we have to restrict the consideration to the control flow, since the equations of the data flow are construction from the guarded actions in a subsequent step. Restriction the consideration to the control flow allows us then to combine the surface and depth parts in a single circuit. To this end, we consider circuits with an interface as shown in Figure 6.4.
The inputs and outputs of these circuits have the following meaning for a statement $S$:

- $E$ represents all wires for inputs and outputs
- $\text{start}$ is the control flow enter condition ($\text{go}^n$)
- $\text{susp}$: suspend the computation (freeze the control)
- $\text{kill}$: abort the computation
- $\text{inst} = \text{inst}_h (S)$
- $\text{insd} = \text{in} (S)$
- $\text{term} = \text{term} (S)$

The hardware circuits for actions and the $\text{nothing}$ statement are shown on the left hand side of Figure 6.5. This circuit has neither gates nor registers, and simply implements the constant control flow predicates $\text{inst} = \text{inst}_h (S) = \text{true}$, $\text{insd} = \text{in} (S) = \text{false}$, and $\text{term} = \text{term} (S) = \text{false}$.

The right hand side of Figure 6.5 shows the hardware circuit of the $\ell:\text{pause}$ statement. As can be seen the transition equation of the D-FlipFlop is $\text{next}(\ell) = \text{start} \lor \text{susp} \land \ell$, which is the result of the disjunctive merge of the surface transition $\text{next}(\ell) = \text{start}$ and the depth transition $\text{next}(\ell) = \text{susp} \land \ell$. The outputs $\text{inst} = \text{inst}_h (\ell:\text{pause}) = \text{false}$, $\text{insd} = \text{in} (\ell:\text{pause}) = \ell$, and $\text{term} = \text{term} (\ell:\text{pause}) = \ell$ are clear.

Figure 6.6 shows the hardware circuits for conditional, sequence, and parallel statements. It can be easily verified that these circuits are implemented as described in the compilation of the surface and depth. The control flow predicates at the outputs are thereby always clear, since they directly follow the previous definitions. We therefore consider the inputs $\text{start}$, $\text{susp}$, and $\text{kill}$ in more detail.

The start conditions of the substatement of a conditional statement are obtained by the if-condition $\sigma$ and the given start condition as described in the surface compilation. The start condition of the second substatement $S_2$ of...
a sequence \( \{S_1; S_2\} \) is the disjunction of \( I_1 \land \text{start} \) and \( T_1 \land \lnot(susp \lor kill) \) which are the start conditions of the surfaces of \( S_2 \) in the surface and depth compilation, respectively. The start condition of the parallel statement is very simple.

The outputs directly follow the definition of the control flow predicates. Note, however, that we make use of the invariants that in if-statements and sequences, at most one of the substatements can be active. Hence, the termination of the if-statement is simply the disjunction of the termination predicates of the substatements, and we do not explicitly demand that the other statement is not active when one of the substatements terminate.

The body of a do-loop is started in the surface when \( \text{start} \) holds, and is started in the depth when \( \lnot(susp \lor kill) \lor \sigma \land T \) holds. The \( susp \) and \( kill \) signals are simple forwarded to the body statement.

The delayed preemption statements are started in the surface when they see a start signal, and there is not additional start in the depth. Clearly, the \( susp \) and \( kill \) signals have to be updated before forwarding them to the body statement. This is done as described in the compilation of the depth. Note here that the suspension and abortion contexts are given different priorities: As we actually compile \( \text{suspend abort } S \text{ when } \sigma \) \text{ when } \text{kill} \text{ when } \text{susp} \), we can simply add the new abortion condition to \( \text{kill} \) as a disjunction according to the following program transformation:

\[
\begin{bmatrix}
\text{suspend} \\
\text{abort} \\
\text{abort} \\
S \\
\text{when}(\sigma) \\
\text{when}(\text{kill}) \\
\text{when}(\text{susp})
\end{bmatrix} \equiv 
\begin{bmatrix}
\text{suspend} \\
\text{abort} \\
S \\
\text{when}(\text{kill} | \sigma) \\
\text{when}(\text{susp})
\end{bmatrix}
\]

Analogously, the suspension context is updated according to the following program transformation:

\[
\begin{bmatrix}
\text{suspend} \\
\text{abort} \\
\text{suspend} \\
S \\
\text{when}(\sigma) \\
\text{when}(\text{kill}) \\
\text{when}(\text{susp})
\end{bmatrix} \equiv 
\begin{bmatrix}
\text{suspend} \\
\text{abort} \\
S \\
\text{when}(\text{kill}) \\
\text{when}(\text{susp} | \lnot\text{kill} \land \sigma)
\end{bmatrix}
\]

We have not listed hardware circuits for immediate preemption statements, which would be, of course, also possible. However, there is the additional difficulty that the preemption condition \( \sigma \) must be evaluated in the surface and in the depth. Clearly, the evaluation in the surface requires to consider the incarnation level function to keep track of possibly reincarnated local variables. The same holds for \text{while} loops.
Finally, note that the listed hardware circuits only implement the control flow, which is also directly computed in form of an equation system by the algorithms of Section 5.2. The data flow must still be computed as given in Section 5.2 by the intermediate step of the guarded actions.

In this chapter, we discuss different compilation methods to translate Quartz programs to equivalent software programs, where 'software' is usually means 'sequential' software, i.e., a program that is executed by a single processor only. Hence, compared to the hardware synthesis of the previous chapter, the compilation techniques of this chapter are the other extreme case for compiling Quartz programs. The techniques considered in this chapter are the software synthesis based on equation systems (Section 6.3), where we consider different evaluation schemes to evaluate the equation systems on single- or multi-processor systems. The first compilation scheme shows up a tight relationship to dataflow computers that have been invented in the eighties to perform efficient parallel computations. Second, we consider the software synthesis based on JobCode that is based on slicing the given Quatz program into small sequential pieces that are triggered by active control flow locations.

6.2 Software Synthesis based on Explicit Extended Finite State Machines

6.3 Software Synthesis based on Equation Systems

In the previous section, we have presented a translation to hardware circuits in detail. In a more abstract sense, this translation is rather a translation to an equation system such that there is exactly one equation for each output, local and location variable (including potential reincarnations of local variables). In every reaction step, new input variables are read and with these new values, all equations of the equation system can be evaluated. Clearly, we have to consider the different dependencies if the equation system has cyclic dependencies.

If the equation system is acyclic, the evaluation is very simple: we can evaluate the different equations in an arbitrary order using one more processors. However, the presence of cyclic dependencies that are moreover data-dependent makes the evaluation of the equation system rather complicated. The potential data-dependency does not allow us to construct a static schedule for the evaluation, so that the evaluation order of the single equations may differ from reaction step to reaction step.

A simple strategy to evaluate the equation system can be obtained from dataflow computers [11, 16, 17, 46]. To this end, we have to endow the used data types with an additional value \( \bot \), which transforms the special case of boolean equation systems to ternary equation systems. The restriction to ternary equation systems yields the special case of ternary evaluation of
6.3 Software Synthesis based on Equation Systems

boolean hardware circuits [57] that is one way to define the set of constructive synchronous programs (it is also equivalent to the must/can analysis of Berry [42]).

Besides the additional value \( \bot \), the next ingredient is the transformation of the equation system to a dataflow graph. This is not really a transformation, since we simply consider the syntax trees of the right hand side expressions which are already our dataflow graphs. The final step is to evaluate this dataflow graph so that nodes are ‘fired’ whenever the require data values for its evaluation are available.

The main problem is thereby the well-known token matching problem: This problem is due to the more general problem to construct a linear time algorithm to evaluate the potentially cyclic equation system. To this end, we should visit every node of the dataflow graph at most a constant number of times, i.e., it is not possible to evaluate the entire equation system as suggested by the Tarski-Knaster fixpoint iteration (see Section ??). Instead, we have to maintain ‘pointers’ to those nodes where the evaluation was not possible since at least one operand was not available. However, also this approach is too expensive, since we could consider the same node several times. A more efficient solution for the evaluation of the equation system is obtained by driving the evaluation by the available data values themselves. Thus, we borrow ideas developed for dataflow computers.

6.3.1 Dataflow Evaluation of Equation Systems

In dataflow computers, the so-called explicit token store architecture has been introduced to efficiently solve the problem. In order to simplify the following explanations, we restrict the discussion to operations with either one or two arguments and to dataflow graphs where nodes have either none, one, or two successor nodes.

The explicit token store architecture maintains tokens in a token queue, where each token \( t \) consists of a value \( t\.val \), a reference to a node in the dataflow graph \( t\.node \) and a boolean flag \( t\.left \) that indicates whether \( t\.val \) is the left hand operand of the corresponding operation. Each node \( n \) of the dataflow graph is endowed with a function \( n\.fun \) and optionally another value \( n\.val \). Moreover, a node \( n \) holds information of its left and right successor nodes \( n\.suc1 \) and \( n\.suc2 \), and whether the edges from \( n \) to \( n\.suc1 \) and from \( n \) to \( n\.suc2 \) refer to the left operand of the target nodes. This is stored in the boolean flags \( n\.left \) and \( n\.right \). Finally, a boolean flag \( n\.bsy \) is used to indicate whether the entry \( n\.val \) holds a valid value.

The execution then consists of the following steps:

- Pick a token \( t \) from the head of the token queue.
- Check whether \( t\.node\.bsy \) holds.
  - If so, we have both operands of our operation, and hence, we can compute a new value. One of the operands, namely \( v_1 := t\.val \), is
obtained from the token \( t \), and the other operand is obtained from the dataflow graph’s node \( v_2 := t.\text{node}.\text{val} \). Depending on whether \( t.\text{left} \) holds or not, we now either compute \( t.\text{node}.\text{fun}(t.\text{val}, t.\text{node}.\text{val}) \) or \( t.\text{node}.\text{fun}(t.\text{node}.\text{val}, t.\text{val}) \), which is our result value \( v \). This computed value \( v \) is used to generate new tokens in the token queue that refer to the successor nodes of \( t.\text{node}.\text{left} \) and \( t.\text{node}.\text{right} \).

- If \( t.\text{node}.\text{bsy} \) does not hold, we only have one operand of the desired operation. Thus, the operation can not be fired, and we have to wait for the second operand. We therefore store the value \( t.\text{val} \) in the value slot \( t.\text{node}.\text{val} \) of \( t.\text{node} \), and set the flag \( t.\text{node}.\text{bsy} \).

A pseudo-code implementation is shown in Figure 6.8. It is important to note that a node of the dataflow graph is only visited when we have an operand for that node. Hence, it follows that we visit each node of the dataflow graph at most twice: namely in those cases, where the node has two operands that appear one after the other. Therefore, the overall evaluation of the dataflow graph is done in a time that is linear by the number of nodes of the dataflow graph, which is essentially the size of the original equation system.

Moreover, inactive nodes, i.e., nodes that do not receive tokens are never visited at all. This is a major distinction to the evaluation as done by a hardware circuit. Using a hardware circuit for the evaluation, we use for each node of the dataflow graph one processor (which is a simple gate) that checks for availability of data all the time. This is reasonable if we have enough processors like in a hardware circuit, but it is not possible if the number of processors is less. In this case, we definitely wish to use the limited number of processors to perform some reasonable computation instead of checking that there is nothing to do.

Before we consider further aspects like multithreaded execution and the relationship to constructive programs, we consider some examples.

### 6.3.2 Example Dataflow Evaluations

- Program P01 leads to the equation system and dataflow graph as shown in Figure 6.9

### 6.3.3 Dataflow Evaluation and Multithreaded Execution

The code given in Figure 6.8 is meant to be executed on a single processor. Clearly, more than one node can usually fire in a dataflow graph. Thus, it would be reasonable to employ several processors for the execution of a dataflow graph. Fortunately, it is straightforward to modify the code for a given number \( N \) of processors. To this end, we simply grab \( N \) tokens from the token queue (if that number of tokens is available) and we process these tokens on the \( N \) processors as shown in Figure 6.8. Note that the insertion
order of the generated tokens into the token queue is irrelevant and that we therefore do not have to care about races. However, we have to care about the situation when a node has currently no operands and the processors pick all of the required operands at once from the token queue. For this reason, we have to make sure that the write access to the nodes in the dataflow graph (which happens when the first operand is written to the graph’s node) is mutually exclusive. If the mutual exclusive access to the nodes in the dataflow graph is guaranteed, then we have no problem for a multithreaded execution of synchronous programs.

6.3.4 Dataflow Evaluation and Constructive Programs

We can establish a close relationship to the constructiveness of programs and the evaluation of dataflow graphs. It is not difficult to see that we can mimic the Tarski-Knaster fixpoint iteration by using sufficiently many processors. Also, with any smaller number of processors we can evaluate the expressions in a corresponding order so that after some steps, one iteration of the fixpoint iteration has been performed. Thus, it follows that the dataflow evaluation scheme can execute every constructive program. However, there is one caveat: Recall that our operators are not strict in the sense of functional programming languages, i.e., they may yield values different to $\bot$ even if one of the arguments is $\bot$. In particular, we have $x \land y = 0$ in case one of the operands is 0, and we $x \lor y = 1$ in case one of the operands is 1. This enables us to perform some sort of lazy evaluation that can proceed even if one of the arguments is missing.

We have to incorporate these rules in the dataflow evaluation machinery in order to establish the desired relationship between dataflow graphs and constructive synchronous programs. This is not very difficult: if a token with the value $\text{false}$ arrives at a $\text{and}$ node, the node can fire even if this is the first argument that arrived. Analogously, if a token with the value $\text{true}$ arrives at a $\text{or}$ node, the node can fire even if this is the first argument that arrived$^1$.

Using these modifications, it is easily seen that the dataflow evaluation scheme can mimic the Tarski-Knaster iteration, and therefore can execute every constructive program correctly. The converse is also easily seen, and therefore we have the following theorem:

**Theorem 6.1 (Constructiveness and Dataflow Evaluation).** The dataflow evaluation of the equation system generated from a synchronous program terminates successfully iff the synchronous program is constructive. A successful termination is thereby a termination of the algorithm of Figure 6.8 (thus an empty token queue) such that all output edges have obtained values.

$^1$ If we would moreover use $\text{ite}$ nodes to implement a conditional case expression, then we could also neglect the then or else branch depending on the truth value that arrives for the condition.
The relationship between the two algorithms can be established by the observation that the additional value $\bot$ that is used in the fixpoint iteration of the causality analysis indicates that there is not yet a value for the further evaluation. It is very important to use non-strict operators here to break dependency cycles as shown in Figures 6.11 and 6.12.

Figure 6.11 shows the Quartz program $P_{13}$ that is constructive for all inputs: The program responds without emissions of outputs. The equation system obviously has a cyclic dependency which does not allow to evaluate the equation system by simply evaluating the equations one after the other. Instead, we have to switch between the equations after we have partially evaluated their right hand sides. The interesting observation is that the truth value of the input $x$ breaks the cycle at runtime: Independent of whether $x$ is true or false, the nonstrictness of the $&$-operator allows one of the nodes on the cycle to fire without waiting on the other argument as can be seen in Figure 6.12. Note that due to the unpatience of the nodes, their second arguments finally arrive, but are never consumed. Hence, it may be the case that tokens remain on inner nodes (but the token queue will always become empty).

For the evaluation of the dataflow graphs, we have to insert tokens with the known values for those variables where we already know the correct value. Absence of a token means that we do not yet know the value of that variable. With this meaning of the tokens, it is a clear consequence that we have to insert tokens with value $\text{true}$ for the inactive locations. However, we do not prefer to move these tokens, since this may result in the evaluation of inactive code which is something we want to avoid.

One might wonder whether it is necessary to insert tokens for the inactive locations at all. In many cases, this is not necessary, since the active locations should determine the correct values of the output variables by triggering the appropriate assignments. However, there are cases where also the information that locations are inactive is required:

- The precondition of the second part $S_2$ of a sequence $\{S_1; S_2\}$ implies the predicate $\text{term}(S_1)$. For parallel statements $\{S_1 \parallel S_2\}$, we have defined $\text{term}(\{S_1 \parallel S_2\})$ as $\text{term}(S_1) \land \neg \text{in}(S_2) \lor \text{term}(S_2) \land \neg \text{in}(S_1) \lor \text{term}(S_1) \land \text{term}(S_2)$, so that negative occurrences of the location variables can occur in the guards $\gamma$ of guarded actions $(\gamma, C)$. The example below shows a program where the information that some locations are currently inactive is really needed for the execution of the actions.
- There are programs where the reaction-to-absence is required to make a further progress in the fixpoint iteration (see Figure 4.8 on page 103). To this end, we have to check that no guard of any of the guarded actions that can modify a variable $x$ are active. This may require to check the corresponding locations are inactive.

The above examples already explain that we have to provide tokens with value $\text{false}$ for the inactive control flow locations. The following program is a concrete example to demonstrate the first one of the above reasons:
module P(event &x) {
    {
        {ℓ₁:pause; ℓ₂:pause;}
        ||
        {ℓ₃:pause; ℓ₄:pause; ℓ₅:pause;}
    }
    emit x;
}

Using the start signal \( st \), we obtain the following guard of the action \( \text{emit } x \):

\[
\ell₂ \land \neg \ell₃ \land \neg \ell₄ \land \neg \ell₅ \lor
\ell₅ \land \neg \ell₁ \land \neg \ell₂ \lor
\ell₂ \land \ell₅
\]

When started, the program will first reach state \( \{ℓ₁, ℓ₃\} \), then state \( \{ℓ₂, ℓ₃\} \), and then the state \( \{ℓ₅\} \). In this state, we definitely provide a token with value true for location \( ℓ₅ \) so that the trigger condition can be reduced by the yet incomplete data flow reduction to \( \neg ℓ₁ \land \neg ℓ₂ \lor ℓ₂ \). If \( ℓ₁ \) and \( ℓ₂ \) are \( \bot \), we can not further evaluate this guard, i.e., we would also have the unknown value \( \bot \) for it, since to token comes out of its root node. The trigger condition becomes true if we provide the two false tokens for the inactive locations \( ℓ₁ \) and \( ℓ₂ \).

A program that requires a reaction-to-absence is the following one:

module Q(event &x) {
    ℓ:pause
    emit x;
}

We obtain the guarded action \((ℓ, \text{emit } x)\). In the initial reaction, we therefore provide the true token for the start condition \( st \), which is however nowhere needed. To find out the we have to determine the value of \( x \) by the reaction-to-absence, we have to check the location \( ℓ \) is currently inactive, so that we have to provide also a token with value false for \( ℓ \).

Hence, we have to provide tokens also for the inactive locations, which is consistent with the meaning of tokens that a token with a value means that we know the value of that variable, and absence of tokens means, we do not know the value, which corresponds with the value \( \bot \).

The evaluation of the data flow graphs should however treat the tokens with different priorities: It seems to be reasonable to first prefer the propagation of the tokens of the active locations, and if these can not be moved further, then the blocked nodes should be used to pull the further evaluation in terms of lazy evaluation. This means that the blocked nodes should request their operands from their child nodes, so that the further evaluation can proceed. These requests will then propagated down the data flow graph until leaf nodes with hopefully known values are reached. Otherwise, we have to propagate some of the inactive location tokens inside due to the reasons that have been explained above.
By the above explanations, it becomes clear that the dataflow evaluation can be very efficient when all variables are given values at every point of time. However, a purely sequential program that only assigns a few output variables per macro step, can behave very bad, since in every reaction step, we require the reaction-to-absence, and therefore must trigger the inactive location tokens for this reason. However, if the inactive location tokens are triggered before the other tokens, there is a good chance that they already generate false tokens for many of the guards in question without completely evaluating the complete guards.

An optimal strategy to efficiently evaluate the expressions is hard to find. In general, we have to distinguish between a demand-driven and a data-driven evaluation [254]. A demand-driven evaluation will ask for the value of the output and local variables, which will trigger new requests to evaluate other subexpressions. In contrast, in a data-driven evaluation strategy available operands will trigger the next evaluation even though the result might not be necessary. There are advantages and disadvantages of both strategies [254].

A possibly good heuristic could be to organize the location variables in a data-driven fashion, while the other nodes should be evaluated by a demand-driven strategy. Once the tokens for the location variables have been propagated inside the graph, the piece of code remains that corresponds with the transition of the extended finite state machine. It is this part of the execution that is the overhead with respect to the code generation that is based on extended finite state machines.

Hence, once we have constructed the dataflow graphs, we can estimate the overhead compared to the automaton code. If the overhead should be too big, we could optimize the dataflow graph in that we make case distinctions on some of the location variables, so that the code size increases, but the reaction time is improved.

### 6.3.5 Adjusting the Granularity by Macro Nodes

The dataflow evaluation of equation system can be significantly more efficient than the simple Tarski-Knaster fixpoint iteration when only a single or only a limited amount of processors is available. However, if the synchronous program contains a large amount of concurrency, it may happen that the dataflow execution is slower than the simple Tarski-Knaster fixpoint iteration due to the additional overhead for maintaining a token queue and the movement of data values.

Fortunately, we can mix different kinds of evaluation strategies by the introduction of macro nodes. We can treat these macro nodes as simple nodes in the dataflow graph that produce a number of output tokens when evaluated with the required input tokens. We can use optimized code for the evaluation of the macro nodes that may not use the dataflow execution style, and may simply contain statically scheduled code.
For this reason, we can generate code that is a mixture of dynamically and statically scheduled code. Clearly, we prefer to generate statically scheduled code, since we do not waste time on the construction of a schedule at runtime.

6.4 Compiling Synchronous Programs into Graph Code

6.5 Compiling Synchronous Programs into Jobs

An convenient way to generate efficient software for uniprocessor and multiprocessor systems can be obtained by compiling the original synchronous program into single jobs. This idea has been initially presented in [231, 232].

In this section, we describe this computation of a set of jobs for a given Quartz program. As already outlined, the overall idea of the proposed code generator is as follows: For each control flow location $\ell$ of the program, a job $S_\ell$ is computed that has to be executed iff the control flow resumes the execution from location $\ell$. Of course, several jobs may have to be executed in one macro step since several locations can be active at once. In addition to the jobs $S_\ell$ that directly correspond to control flow locations, we additionally use further jobs $S_\lambda$ to avoid code duplication. As explained below, these additional jobs $S_\lambda$ correspond to barriers with a lock variable $\lambda$.

The overall execution of the jobs is described in more detail in the next section. In this section, we describe first the syntax and semantics of the jobs and their computation.

In principle, a job $S_\ell$ consists of a set of guarded actions to implement the data flow of the program and a set of guarded $schedule(\ell)$ statements to implement the control flow of the program. However, we do not compute simple sets of guarded actions and guarded $schedule(\ell)$ statements. Instead, we additionally use further statements like conditional and sequential statements to allow sharing of common conditions. Moreover, we use statements for barrier synchronization to implement the concurrency of synchronous programs. In the following, we discuss these statements in detail.

Definition 6.2. [Job Language] The set of JobCode statements is the smallest set that satisfies the following rules, provided that $S$, $S_1$, and $S_2$ are also Job statements, $\ell$ is a location variable, $x$ is an event variable, $y$ is a state variable, $\sigma$ is a Boolean expression, and $\lambda$ is a lock variable (integer type):

- nothing (empty statement)
- $y = \tau$ and $next(y) = \tau$ (assignments)
- $init(x)$ (initialize local variable)
- $schedule(\ell)$ (resumption at next reaction)
- $reset(\lambda)$ (reset a barrier variable)
- $fork(\lambda)$ (immediately fork to job $\lambda$)
- $barrier(\lambda, c)$ (try to pass barrier $\lambda$)
if(σ) $S_1$ else $S_2$ (conditional)

$S_1; S_2$ (sequential composition)

Note that there is no longer a parallel statement and also the abort/suspend statements are no longer required. Moreover, there are no loops, since we can implement them by the help of schedule statements (explained below). Moreover, all job statements are instantaneous\(^2\).

The atomic statements nothing, emit $x$, emit next($x$), $y = \tau$, and next($y$) = $\tau$ have the same meaning as in Quartz programs. The meaning of conditionals and sequences is also the same as in Quartz. The statement init($x$) replaces a local variable declaration as follows: when executed, it first removes $x$ from the current context as well as pending (delayed) assignments to $x$, and then gives $x$ the initial default value according to the type of $x$.

The schedule($\ell$) statement corresponds with a ‘delayed goto’ to a control flow location $\ell$ of the Quartz program. When executed, it simply puts the label $\ell$ into a schedule, so that the runtime environment will execute the corresponding job $S_\ell$ in the next reaction step. Note, however, that schedule($\ell$) is instantaneous, so that schedule($\ell_1$); schedule($\ell_2$) will put both $\ell_1$ and $\ell_2$ at once into the schedule for the next reaction step.

The statements reset($\lambda$), join($\lambda$), and barrier($\lambda, c$) are used to implement concurrency based on barrier synchronization. The statement barrier($\lambda, c$) declares a barrier with an integer lock variable $\lambda$ and an integer constant $c$ as threshold. Executing this statement checks whether $\lambda \geq c$ holds, and if so, it immediately terminates, so that a further statement $S$ can be executed in a sequence barrier($\lambda, c$); $S$. If $\lambda < c$ holds, the execution stops, so that this control thread terminates.

Executing reset($\lambda$) simply resets $\lambda = 0$, and join($\lambda$) first increments $\lambda$ and then executes a job $S_\lambda$ that is associated with the barrier whose lock variable is $\lambda$. Usually (and in our compiled jobs always), it is the case that the code of job $S_\lambda$ is a sequence barrier($\lambda, c$); $S'_\lambda$ with some job statement $S'_\lambda$.

Using the statements for barrier synchronization, it is straightforward to execute parallel code on a uniprocessor machine: We associate with each parallel statement a barrier with lock variable $\lambda$ and threshold $c = 2$ that is reset when the parallel statement is started. When a thread of the parallel statement terminates, it executes a join($\lambda$) statement. If both threads have executed their final join($\lambda$) statements, the barrier will be passed, so that the code $S'_\lambda$ following the associated barrier($\lambda, c$) statement in the job $S_\lambda$ associated with the barrier can be executed.

The implementation of the barrier synchronization for other architectures may (and must) be different. Hence, it depends on the platform that is used to execute the program, while our jobs remain architecture-independent. Different implementations for barrier synchronization already exist \cite{8} for hard-

\(^2\)The job language is therefore also a synchronous language that is however not meant to be offered to the programmer. Instead, it is used as an intermediate language \cite{197} that could, in principle, be the target for many synchronous languages.
ware, software on multiprocessors, and software on uniprocessors, so that our jobs can be executed for all of these platforms.

Figure ?? shows the algorithms to compile a given Quartz statement into a preemption tree whose nodes are labeled with a set of jobs and a preemption condition (except for the root node). Such preemption trees are defined below and are the representation of a program in terms of job code.

**Definition 6.3. [Preemption Trees]** Given a set $\mathcal{J}$ of pairs $(\lambda, S_\lambda)$ with job code statements $\lambda$ that have the unique names $\lambda$, and a boolean expression $\sigma$, then the following are preemption trees, provided that $\lambda$ is the name of a job:

- $PmtRoot(T, \mathcal{J}, \lambda)$
- $WeakAbort(T, \mathcal{J}, \sigma, \lambda)$
- $StrongAbort(T, \mathcal{J}, \sigma, \lambda)$
- $WeakSuspend(T, \mathcal{J}, \sigma)$
- $StrongSuspend(T, \mathcal{J}, \sigma)$

Preemption trees contain the jobs that are computed for a given Quartz program in the job sets $\mathcal{J}$ of their nodes. Each node of the preemption tree except for the root node uniquely corresponds with a preemption statement and the nesting of the preemption statements is given by the structure of the preemption tree. The jobs of the root node do not stem from the body of a preemption statement, while the other jobs stem from the body of the preemption statement that corresponds with the node of the preemption tree.

In the root node $PmtRoot(T, \mathcal{J}, \lambda)$, the job $\lambda$ is the initial job that has to be executed when the entire program is started (the surface of the program). In case of an abortion node, the job $\lambda$ is the job that is executed when the abortion takes place. Note that in this case, the jobs of the abortion statement's body are not allowed to schedule jobs (neither in the schedule of the current nor of the next macro step).

### 6.6 Formal Semantics of the Job Code

In the following, we present a formal semantics of the job code that is also the specification of an interpreter to execute job code. Similar to the semantics of the Quartz language, we present the semantics of the Job code in an operational form that can be formalized by SOS rules that directly specify an interpreter algorithm.

Similar to the semantics of the Quartz language, we define the semantics of the job code language in two steps: The first step is used to compute the current outputs to given inputs based on a must/can analysis very similar to that of the Quartz language. As expected, the second part of the semantics will then compute from the completed environment the set of delayed actions and the set of jobs that have to be executed in the next macro step.
6.6.1 Approximating the Environment

The first part of the semantics is given by SOS rules that determine the current environment by a must/can analysis as well as the set of jobs that can/must be executed in the current macrostep. These rules are of the following form

$$(E, h, S) \rightarrow (E', h', D_{\text{must}}, D_{\text{can}}, J_{\text{must}}, J_{\text{can}}, t)$$

The intuitive meaning of these rules is as follows: Clearly, $E$, $h$ describe the current environment where the job statement $S$ is started. Due to its execution, the environment is immediately updated to $E'$, $h'$, which is different to the SOS rules of Quartz. The reason for the immediate update of $E$ is that the barrier-statements of the job code language immediately increase the lock variable.

The sets $D_{\text{must}}$ and $D_{\text{can}}$ have the same meaning as described in the corresponding rules for the Quartz language: both sets consist of pairs $(\alpha, h)$, where $\alpha$ is an immediate action and $h$ is the incarnation level function that has to be used for the execution of the action $\alpha$. Clearly, $D_{\text{must}}$ contains the actions that must be executed while $D_{\text{can}}$ contains the actions that can be executed in the current environment. In a similar way, we compute sets $J_{\text{must}}$ and $J_{\text{can}}$ that are the set of jobs that must be and can be executed in the current macrostep. Finally, $t$ is a boolean flag that denotes if the execution failed to pass a barrier.

The definition of the SOS rules are given in Figure 6.15. As can be seen, most rules are deterministic in the sense that they do not introduce uncertainty in that the must/can parts would differ. The only exception is the if-else statement in the case where the condition $\sigma$ evaluates to an unknown value.

Note that the environment is modified during the ‘execution’ of the SOS rules. However, the only modification is due to the incrementation of the barriers when a \textit{barrier} is executed. In all other cases, the environment is not modified. The incarnation level function is also updated during the ‘execution’ of the SOS rules when a \textit{init} is executed. For this statement, we distinguish between two cases: in the first case there are already enough reincarnations in the current environment, so that we only have to adjust the current incarnation level function. In the second case, we have to additionally add another incarnation to $E$ that initially receives the unknown value $\bot$.

In the other cases, the environment is not modified (except for taking the intersection/maximum of previously computed environments). The assignment statements only collect the assignments together with the current incarnation level function so that the expressions can be evaluated later on.

The \textit{schedule} simply has no effect, since we are not yet interested in the next macro step. The \textit{fork} statement simply adds a job to the current job schedule. The barrier statement first increments the current value of the barrier variable, and then tests whether the threshold has been reached or not. The result of this test is the flag for testing the failure of threshold passing.
The effect of this flag is seen by the rules for sequences: In the first case, the execution of $S_1$ fails to pass a barrier. Hence, we do not execute $S_2$ and only forward the results of the execution of $S_1$. In the second rule, $S_1$ passes all barriers, so that the execution reaches $S_2$. The results are then the combinations of the two executions.

Finally, we have to explain the rules for the conditional statement. In case the condition $\sigma$ can already be evaluated to a boolean value, we either execute the ‘then’ or the ‘else’ branch to obtain the desired results. Otherwise, we execute both the ‘then’ and the ‘else’ branch and combine the results appropriately, i.e., taking the intersections in the ‘must’ case and the unions in the ‘can’ case. A crucial point is the final result of the barrier passing flag.

We need additional rules to execute a set of jobs and also to execute preemption trees. The execution of a set of jobs is straightforward: We select an arbitrary job out of the set and execute it according to the rules given in Figure 6.17. After this, we add the jobs that are immediately scheduled by this execution to the currently scheduled jobs, and repeat the execution as long as jobs are available.

### 6.6.2 Preparing the Next Macro Step

Having completed the current environment by appropriate values for the local and output variables, we are now ready to compute the information we need to execute the next macro step. To this end, we use again SOS rules that are this time of the following form:

$$\langle E, h, S \rangle \rightarrow_j \langle E', h', J_{\text{now}}, J_{\text{next}}, D, t \rangle$$

The meaning of these rules is as follows: if the job statement $S$ is executed in the current environment $(E, h)$ then it immediately schedules the jobs $J_{\text{now}}$ for execution in the current macro step and it schedules the jobs $J_{\text{next}}$ for execution in the next macrostep. Moreover, $D$ consists of a list of pairs $(y, v)$ where $y$ is a variable that has been assigned the value $v$ by a delayed assignment of $S$. Analogously to the semantics of the Quartz language, the assignment of $v$ to $y$ has to be performed in the next macrostep, i.e., before the next macrostep starts. Clearly, $(E', h')$ is the updated environment that differs from $E$ only in that barrier variables may have been increased and some local declarations have been executed (which increased the levels of $h$ to those in $h'$). Finally, the boolean flag $t$ holds iff the execution failed to pass a barrier statement.

The rules are given in Figure 6.17 and can be intuitively understood. Note that immediate assignments only check that the current environment is consistent with their immediate effect, and that the execution of delayed assignments leads to entries in the set of delayed actions $D$ that will be used to prepare the environment of the next macro step. Moreover, note that we only have one rule for $\text{init}$, where it is assumed that the environment has enough incarnations of the considered local variable. The rules for the other statements are simple.
Fig. 6.6. Hardware Circuits for Conditional, Sequence, and Parallel Statements

Fig. 6.7. Hardware Circuits for Loops and Delayed Preemption Statements
class nodeDF {
    opDF fun;
    int narg;
    valueDF val;
    bool bsy;
    nodeDF suc1;
    bool left1;
    nodeDF suc2;
    bool left2;
}

class tokenDF {
    nodeDF node;
    valueDF val;
    bool left;
    tokenDF next;
}

void InsertToken(nodeDF n,valueDF v,bool l,tokenDF tend) {
    tnew = new tokenDF();
    tnew.node = n;
    tnew.val = v;
    tnew.left = l;
    tnew.next = 0;
    tend = tnew;
}

void evaluateDF(tokenDF tstart,tend) {
    nodeDF n;
    tokenDF tnew;
    valueDF v;
    bool isLeft;
    while(t!=0) {
        // pick a token from the token queue
        n = tstart.node;
        v = tstart.val;
        isLeft = t.left;
        t = t.next;
        // evaluate if possible
        if(nargs==1) {
            v = compute(n.fun,v);
            if(n.suc1) InsertToken(n.suc1,v,n.left1,tend);
            if(n.suc2) InsertToken(n.suc2,v,n.left2,tend);
        } else if(n.args==2)&n.bsy) {
            v = isLeft?compute(n.fun,v,n.val):compute(n.fun,n.val,v);
            if(n.suc1) InsertToken(n.suc1,v,n.left1,tend);
            if(n.suc2) InsertToken(n.suc2,v,n.left2,tend);
        } else { // store value
            n.bsy = true;
            n.val = v;
        }
    }
}

Fig. 6.8. Dataflow Evaluation of Equation Systems
module P01(event x, &y1, &y2, &y3) {
    if(x) emit(y1); ||
    if(!y1) emit(y2); ||
    if(y2) emit(y3);
}

\[
\begin{align*}
    y_1 &= st \& x \\
    y_2 &= st \& !y_1 \\
    y_3 &= st \& y_2
\end{align*}
\]

\textbf{Fig. 6.9.} Dataflow Graph of Program P01.

\textbf{Fig. 6.10.} Dataflow Evaluation of Program P01.

module P13(event x, &y1, &y2) {
    if(x) {
        if(y1) emit y2; \{ y_1 = st \& !x \& y_2 \}
        else
            if(y2) emit y1; \{ y_2 = st \& x \& y_1 \}
    }
}

\textbf{Fig. 6.11.} Dataflow Graph of Program P13.

\textbf{Fig. 6.12.} Dataflow Graph of Program P13: Despite the cyclic dependencies, the dataflow graph terminates successfully for all inputs.
function Jobs(S, S_n, j)
  case S of
    nothing:
      if S_n = nothing then
        return ([fork(j)], \{\}, \{\})
      else
        return Jobs(S_n, nothing, j)
    emit x, emit next(x), y = \tau, next(y) = \tau:
      (S^o, T, J) = Jobs(S_n, nothing, j);
      return ([S; S^o], T, J)
  \ell : pause:
    (S, T, J) = Jobs(S_n, nothing, j);
    return ([schedule(\ell)]; T, ((\ell, S_i) \cup J)
  if (\sigma) S_1 else S_2:
    \lambda = NewVar();
    (S^1_1, T_1, J_1) = Jobs(S_1, nothing, \lambda);
    (S^2_2, T_2, J_2) = Jobs(S_2, nothing, \lambda);
    (S^3_3, T_3, J_3) = Jobs(S_n, nothing, j);
    S_0 = [if(\sigma) S^1_1 else S^2_2];
    J = \{(\lambda, S^0_0) \cup J_1 \cup J_2 \cup J_3;
    return (S_0, T_1 \cup T_2 \cup T_3, J);
  S_1; S_2 :
    return Jobs(S_1, (S_2; S_0), j)
if (\sigma) S_1 then
  return ParallelJobs(S_1, S_2, S_n, j)
do S while (\sigma):
  return LoopJobs(\sigma, S, S_n, j)
abort S when (\sigma):
  return AbortJobs(S, \sigma, 0, 0, S_n, j);
weak abort S when (\sigma):
  return AbortJobs(S, \sigma, 1, 0, S_n, j);
immediate abort S when (\sigma):
  return AbortJobs(S, \sigma, 0, 1, S_n, j);
weak immediate abort S when (\sigma):
  return AbortJobs(S, \sigma, 1, 1, S_n, j);
suspend S when (\sigma):
  return SuspendJobs(S, \sigma, 0, 0, S_n, j);
weak suspend S when (\sigma):
  return SuspendJobs(S, \sigma, 1, 0, S_n, j);
\ell : immediate suspend S when (\sigma):
  return SuspendJobs(\ell, S, \sigma, 0, 1, S_n, j);
\ell : weak immediate suspend S when (\sigma):
  return SuspendJobs(\ell, S, \sigma, 1, 1, S_n, j);
\{ \alpha x; S \}:
  (S^o, T, J) = Jobs(\alpha x; S^o, T, J);
end case
end

function ParallelJobs(\sigma, S, S_n, j)
  \lambda = NewVar();
  (S^1_1, T_1, J_1) = Jobs(S_1, nothing, \lambda);
  (S^2_2, T_2, J_2) = Jobs(S_2, nothing, \lambda);
  (S^3_3, T_3, J_3) = Jobs(S_n, nothing, j);
  S_0 = [reset(\lambda); S^1_1; S^2_2];
  J = \{(\lambda, [\text{barrier}(\lambda, 2); S^3_3]) \cup J_1 \cup J_2 \cup J_3);
  return (S_0, T_1 \cup T_2 \cup T_3, J);
end

function LoopJobs(\sigma, S, S_n, j)
  \lambda = NewVar();
  (S^1_1, T_1, J_1) = Jobs(S_1, nothing, \lambda);
  (S^2_2, T_2, J_2) = Jobs(S_2, nothing, \lambda);
  S_{\text{reenter}} = [if(\sigma) S^1_1 else S^2_2];
  J = \{(\lambda, S_{\text{reenter}}) \cup J_1 \cup J_2;
  return (S_0, \{P\} \cup T_1 \cup T_2, J);
end

function AbortJobs(S, \sigma, wk, im, S_n, j)
  \lambda_1 = NewVar(); \lambda_2 = NewVar();
  (S^1_1, T_1, J_1) = Jobs(S_1, nothing, \lambda_1);
  (S^2_2, T_2, J_2) = Jobs(S_2, nothing, \lambda_2);
  if im then
    if wk then
      \lambda_0 = NewVar();
      S_{\sigma} = [fork(\lambda_0)];
      J_1 = \{(\lambda_0, S^0_0) \cup J_1;
      else
        S_{\sigma} = [if(\sigma) fork(\lambda_2); else S^1_1];
        else
          S_{\sigma} = [S^1_1];
          J = \{(\lambda_1, [if(\neg\sigma) fork(\lambda_2)]; \cup J_2;
          if wk then \sigma = WeakAbort(T_1, J_1, \sigma, \lambda_2)
          else \sigma = StrongAbort(T_1, J_1, \sigma, \lambda_2)
          return (S_0, \{P\} \cup T_2, J);
      \end
end

function SuspendJobs(\ell, S, \sigma, wk, im, S_n, j)
  \lambda_1 = NewVar(); \lambda_2 = NewVar();
  (S^1_1, T_1, J_1) = Jobs(S_1, nothing, \lambda_1);
  (S^2_2, T_2, J_2) = Jobs(S_2, nothing, \lambda_2);
  if im then
    if wk then
      \lambda_0 = NewVar();
      S_{\sigma} = [fork(\lambda_0)];
      J_1 = \{(\ell, S^0_0) \cup J_1;
      else
        S_{\sigma} = [if(\sigma) schedule(\ell); else S^1_1];
        else
          S_{\sigma} = [S^1_1];
          J = \{(\lambda_1, [if(\neg\sigma) fork(\lambda_2)]; \cup J_2;
          if wk then \sigma = WeakAbort(T_1, J_1, \sigma, \lambda_2)
          else \sigma = StrongAbort(T_1, J_1, \sigma, \lambda_2)
          return (S_0, \{P\} \cup T_2, J);
      \end
end

function jobs from Fig. 6.13.
function CompileJobs(S, j)
    case S of
        nothing : return (fork(j), {}, {});
        y = τ : return ((y = τ; fork(j)), {}, {});
        next(y) = τ : return ((next(y) = τ; fork(j)), {}, {});
        ℓ : return(schedule(ℓ), {}, {});
        if(σ) S1 else S2 :
            (S_1^o, T_1, J_1) = CompileJobs(S_1, j);
            (S_2^o, T_2, J_2) = CompileJobs(S_2, j);
            S_0 = [if(σ) S_1^o else S_2^o];
            return[(S_0, T_1 ∪ T_2, J_1 ∪ J_2);
        S_1 ∥ S_2 :
            λ = NewVar();
            (S_1^o, T_1, J_1) = CompileJobs(S_1, λ);
            (S_2^o, T_2, J_2) = CompileJobs(S_2, λ);
            S_0 = [reset(λ); S_1^o; S_2^o];
            J = {(λ, barrier(λ, 2))} ∪ J_1 ∪ J_2;
            return[(S_0, T_1 ∪ T_2, J);
        do S while(σ) :
            return LoopCompileJobs(σ, S, j)
        abort S when(σ) :
            return AbortCompileJobs(S, σ, 0, 0, j);
        weak abort S when(σ) :
            return AbortCompileJobs(S, σ, 1, 0, j);
        immediate abort S when(σ) :
            return AbortCompileJobs(S, σ, 0, 1, j);
        weak immediate abort S when(σ) :
            return AbortCompileJobs(S, σ, 1, 1, j);
        suspend S when(σ) :
            return SuspendCompileJobs(S, σ, 0, 0, j);
        weak suspend S when(σ) :
            return SuspendCompileJobs(S, σ, 1, 0, j);
        ℓ : immediate suspend S when(σ) :
            return SuspendCompileJobs(ℓ, S, σ, 0, 1, j);
        ℓ : weak immediate suspend S when(σ) :
            return SuspendCompileJobs(ℓ, S, σ, 1, 1, j);
        {α x; S} :
            (S^o, T, J) = CompileJobs(S, nothing);
            return [(init(x); S^o], T, J);
    end case
end

function LoopCompileJobs(σ, S, j)
    λ = NewVar();
    (S_1^o, T_1, J_1) = CompileJobs(S, λ);
    S_reenter = {if(σ) S_1^o else fork(j)};
    J = {(λ, S_reenter)} ∪ J;
    return (S_1^o, T_1, J);
end

function AbortCompileJobs(S, σ, wk, im, j)
    λ_1 = NewVar();
    (S_1^o, T_1, J_1) = CompileJobs(S, λ_1);
    if im then
        S_0 = fork(λ);
        J_1 = {(λ_0, S_1^o)} ∪ J_1;
        else S_0 = S_1^o;
        J = {(λ_1, if(¬σ) fork(j))};
    if wk then T = {WeakAbort(T_1, J_1, σ, j)}
    else T = {StrongAbort(T_1, J_1, σ, j)}
    return(S_0, T, J);
end

function SuspendCompileJobs(ℓ, S, σ, wk, im, j)
    λ_1 = NewVar();
    (S_1^o, T_1, J_1) = CompileJobs(S, λ_1);
    if im then
        S_0 = fork(ℓ);
        J_1 = {(ℓ, S_1^o)} ∪ J_1;
        else S_0 = S_1^o;
        J = {(λ_1, if(¬σ) fork(j))};
    if wk then T = {WeakSuspend(T_1, J_1, σ, j)}
    else T = {StrongSuspend(T_1, J_1, σ, j)}
    return(S_0, T, J);
end

Fig. 6.14. Another Algorithm to Compile Jobs
\[ (E, h, \text{nothing}) \vdash_J (E, h, \{\}, \{\}, \{\}, \text{false}) \]
\[ (E, h, y = \tau) \vdash_J (E, h, \{(y = \tau, h)\}, \{(y = \tau, h)\}, \{\}, \{\}, \text{false}) \]
\[ (E, h, \text{next}(y) = \tau) \vdash_J (E, h, \{\text{next}(y) = \tau, h\}, \{(\text{next}(y) = \tau, h)\}, \{\}, \{\}, \text{false}) \]
\[
E(x) = [v_0, \ldots, v_n] \quad h(x) \leq n
\]
\[
(E, h, \text{init}(x)) \vdash_J (E, h, h^{[x,h(x)+1]}, \{\}, \{\}, \text{false})
\]
\[
E(x) = [v_0, \ldots, v_n] \quad h(x) = n
\]
\[
(E, h, \text{init}(x)) \vdash_J (E, h, \{\}, \{\}, \{\}, \text{false})
\]
\[
(E, h, \text{schedule}(\ell)) \vdash_J (E, h, \{\}, \{\}, \{\}, \text{false})
\]
\[
(E, h, \text{fork}(\lambda)) \vdash_J (E, h, \{\lambda\}, \{\lambda\}, \{\}, \text{false})
\]
\[
(E, h, \text{reset}(\lambda)) \vdash_J (E, h, \{\}, \{\}, \{\}, \text{false})
\]
\[
E(\lambda) = [n]
\]
\[
(E, h, \text{barrier}(\lambda, c)) \vdash_J (E, h, [n, c), \{\}, \{\}, \{\}, n+1 < c)
\]
\[
(E, h, S_1) \vdash_J (E, h, J^1_{\text{must}}, J^1_{\text{can}}, J^1_{\text{true}})
\]
\[
(E, h, S_1) \vdash_J (E, h, J^1_{\text{must}}, J^1_{\text{can}}, J^1_{\text{true}}) 
\]
\[
[E, h, \{\}, \{\}, \{\}, \text{false}]
\]
\[
(E, h, \text{if}^\sigma (S_1 \text{ else } S_2)) \vdash_J (E, h, \{\}, \{\}, \{\}, \{\}, \text{false})
\]
\[
(E, h, \text{if}^\sigma (S_1 \text{ else } S_2)) \vdash_J (E, h, \{\}, \{\}, \{\}, \{\}, \text{false})
\]
\[
\text{Fig. 6.15. Causality Analysis for the Job Language}
\]
function ExecJobList(E, h, Jcur, Jnode)
    Jnow = Jcur \ Jnode;
    Jcur = Jcur \ Jnode \ Dmu;
    Dmu = Dcan = {};
    while (Jcur ≠ {}) do
        i = select(Jcur);
        Jcur = Jcur \ {i};
        (E, h, Dmu, Dcan, Jmu, Jcan, Jcan, t) = ρ3 (E, h, S_i);
        Jnow = Jnow ⊔ (Jnow \ Jnode);
        Jcur = Jcur ⊔ (Jcur \ Jnode);
        Jnext = Jnext ⊔ Jnow \ Jnode;
        D = D ∪ D'; C = C ∪ C';
    end
    // we have Jnow ⊔ Jnode = {};
    return (E, h, Jnow, Jnext, D, C)
end

function JobsInTree(Ψ)
case Ψ of
    PmtRoot(T, J, λ_0):
        L = Uₜ∈T JobsInTree(P);
        return L ∪ \{i | ∃S_i, (i, S_i) ∈ J\};
    StrongAbort(T, J, σ, λ):
        L = Uₜ∈T JobsInTree(P);
        return L ∪ \{i | ∃S_i, (i, S_i) ∈ J\};
    WeakAbort(T, J, σ, λ):
        L = Uₜ∈T JobsInTree(P);
        return L ∪ \{i | ∃S_i, (i, S_i) ∈ J\};
    StrongSuspend(T, J, σ):
        L = Uₜ∈T JobsInTree(P);
        return L ∪ \{i | ∃S_i, (i, S_i) ∈ J\};
    WeakSuspend(T, J, σ):
        L = Uₜ∈T JobsInTree(P);
        return L ∪ \{i | ∃S_i, (i, S_i) ∈ J\};
end

function ExecTree(E, h, Jcur, J)
    3 = JobsInTree(J);
    if Jcur ∩ 3 = {} then return (E, h, Jcur, {}, {}, {}); end;
    case 3 of
    Root(T, Jnode):
        do
            (E, h, Jnow, Jnext, D, C) = ExecJobList(E, h, Jcur, Jnode);
            for T ∈ T do
                (E, h, Jnow, Jnext, D', C') = ExecTree(E, h, Jnow, T);
                Jnext = Jnext ∪ Jnow ∩ Jnode;
                D = D ∪ D'; C = C ∪ C';
            end;
            while (Jnow ∩ Jnode ≠ {});
            return (E, h, Jnow, Jnext, D, C)
        end
    StrongAbort(T, σ, λabort, Jnode):
        case [σ]ₜ of
            true: return return (E, h, {λabort} ∪ (Jcur \ 3), {}, {}, {});
            ⊥: return (E, h, Jcur \ 3, D, C) = ExecTree(E, h, Jcur, Root(T, Jnode));
        end
    WeakAbort(T, σ, λabort, Jnode):
        case [σ]ₜ of
            true: return return (E, h, {λabort} ∪ Jnow, {}, D, C);
            ⊥: return (E, h, Jcur \ 3, D, C) = ExecTree(E, h, Jcur, Root(T, Jnode));
        end
    StrongSuspend(T, σ, λAbort, Jnode):
        case [σ]ₜ of
            return return (E, h, Jcur \ 3, Jcur ∩ J, {}, {}, {});
            ⊥: return (E, h, Jcur \ 3, D, C) = ExecTree(E, h, Jcur, Root(T, Jnode));
        end
    WeakSuspend(T, σ, λSuspend, Jnode):
        (E, h, Jnow, Jnext, D, C) = ExecTree(E, h, Jcur, Root(T, Jnode));
        case [σ]ₜ of
            return return (E, h, Jcur \ 3, Jcur ∩ J, {}, {}, {});
            ⊥: return (E, h, Jcur \ 3, D, C) = ExecTree(E, h, Jcur, Root(T, Jnode));
        end
end

Fig. 6.16. Computing the Outputs of an Emptition Tree
\[ (E, h, \text{nothing}) \rightarrow_j (E, h, \{\}, \{\}, \text{false}) \]

\[ \| y \|^h = \| \tau \|^h \]

\[ (E, h, y = \tau) \rightarrow_j (E, h, \{\}, \{\}, \text{false}) \]

\[ (E, h, \text{next}(y) = \tau) \rightarrow_j (E, h, \{\}, \{(y, \| \tau \|^h)\}, \text{false}) \]

\[ E(x) = [v_0, \ldots, v_n] \quad h(x) \leq n \]

\[ (E, h, \text{init}(x)) \rightarrow_j (E, [h^{(x)}_x]^{h(x) + 1}, \{\}, \{\}, \text{false}) \]

\[ (E, h, \text{schedule}(\ell)) \rightarrow_j (E, h, \{\ell\}, \{\}, \text{false}) \]

\[ (E, h, \text{fork}(\lambda)) \rightarrow_j (E, h, \{\lambda\}, \{\}, \text{false}) \]

\[ (E, h, \text{reset}(\lambda)) \rightarrow_j ([E]^{(\lambda, h)}_{(0, 0)}, h, \{\}, \{\}, \text{false}) \]

\[ (E, h, \text{barrier}(\lambda, c)) \rightarrow_j ([E]^{(\lambda, h)}_{(0, 0)}, h, \{\}, \{\}, \{\lambda\} + 1 < c) \]

\[ (E, h, S_1) \rightarrow_j (E_1, h_1, J_{\text{now}}, J_{\text{next}}, D_1, \text{true}) \]

\[ (E, h, S_1; S_2) \rightarrow_j (E_1, h_1, J_{\text{now}} \cup J_{\text{next}} \cup J_{\text{now}} \cup J_{\text{next}} \cup J_{\text{now}} \cup J_{\text{next}} \cup D_1 \cup D_2, t_2) \]

\[ \sigma^h = \text{true} \]

\[ (E, h, \text{if}(\sigma) S_1 \text{ else } S_2) \rightarrow_j (E_1, h_1, J_{\text{now}} \cup J_{\text{next}} \cup D_1, t_2) \]

\[ \sigma^h = \text{false} \]

\[ (E, h, \text{if}(\sigma) S_1 \text{ else } S_2) \rightarrow_j (E_1, h_1, J_{\text{now}} \cup J_{\text{next}} \cup D_1, t_2) \]

\[ \text{Fig. 6.17. SOS Rules for the Job Language} \]
function ExecJobList(Δ, h, Jcur, Jnode)
    Jnow = Jcur \ Jnode;
    Jcur = Jcur \ Jnode;
    Jnext = D = \{
    while (Jcur \= \{
        i = select(Jcur);
        Jcur = Jcur \ \{i\};
        (E, h, Jnow, Jnext, D', t_i) \= \{Δ, h, S_i\};
        Jnow = Jnow \cup (Jnow \ Jnode);
        Jcur = Jcur \cup (Jcur \ Jnode);
        Jnext = Jnext \cup Jnext;
        D = D \cup D';
    end
    we have Jnow \ Jnode = \{
    return (E, h, Jnow, Jnext, D)
end

function JobsInTree(Ψ)
    case Ψ of
        PmtRoot(T, J, λ_0):
            L = \bigcup_{P \in T} JobsInTree(P);
            return L \cup \{i | ∃ S_i. (i, S_i) \in J\};
        StrongAbort(T, J, σ, λ):
            L = \bigcup_{P \in T} JobsInTree(P);
            return L \cup \{i | ∃ S_i. (i, S_i) \in J\};
        WeakAbort(T, J, σ):
            L = \bigcup_{P \in T} JobsInTree(P);
            return L \cup \{i | ∃ S_i. (i, S_i) \in J\};
        StrongSuspend(T, J, σ):
            L = \bigcup_{P \in T} JobsInTree(P);
            return L \cup \{i | ∃ S_i. (i, S_i) \in J\};
        WeakSuspend(T, J, σ):
            L = \bigcup_{P \in T} JobsInTree(P);
            return L \cup \{i | ∃ S_i. (i, S_i) \in J\};
    end
end

function ExecTree(Δ, h, Jcur, Ψ)
    J = JobsInTree(Ψ);
    if Jcur \ J = \{
        return (Δ, h, Jcur, \{}, \{}\)
    end
    case Ψ of
        PmtRoot(T, J, λ_0):
            do
                (E, h, Jnow, Jnext, D) = ExecJobList(Δ, h, Jcur, Jnode);
                for T \in T do
                    (Δ, h, Jnow, Jnext, D') = ExecTree(Δ, h, Jcur, T);
                    Jnext = Jnext \cup Jnext';
                    D = D \cup D';
                end
                while (Jnow \ J \= \{
            return (Δ, h, Jnow, Jnext, D)
        StrongAbort(T, J, λ_{abort}, Jnode):
            Ψ = PmtRoot(T, Jnode, 0);
            if [σ]_L \= \{
                return (Δ, h, \{λ_{abort}\} \cup (Jcur \ J), \{}, \{}\)
            else
                return ExecTree(Δ, h, Jcur, Ψ');
            end
        WeakAbort(T, J, λ_{abort}, Jnode):
            Ψ = PmtRoot(T, Jnode, 0);
            if [σ]_L \= \{
                return (Δ, h, \{λ_{abort}\} \cup Jnow, \{}, \{}\)
            else
                return ExecTree(Δ, h, Jcur, Ψ');
            end
        StrongSuspend(T, J, λ_{susp}, Jnode):
            Ψ = PmtRoot(T, Jnode, 0);
            if [σ]_L \= \{
                return (Δ, h, Jcur \ J, Jcur \ J, \{}, \{}\)
            else
                return ExecTree(Δ, h, Jcur, Ψ');
            end
        WeakSuspend(T, J, λ_{susp}, Jnode):
            Ψ = PmtRoot(T, Jnode, 0);
            (Δ, h, Jnow, Jnext, D) = ExecTree(Δ, h, Jcur, Ψ');
            if [σ]_L \= \{
                return (Δ, h, Jcur \ J, Jcur \ J, \{}, \{}\)
            else
                return ExecTree(Δ, h, Jcur, Ψ');
            end
end

Fig. 6.18. Executing Jobs within a Preemption Tree
function MacroStep(ε prv, ℋ prv, D prv, T, J)

// preparation phase
ε new = ReadInputs();
ε new = ε new ∪ \{ (y, 0) | y \in V_{out} \cup V_{loc} \cup V_{bar} \};
ℏ new = \{ (y, 0) | y \in V_{in} \cup V_{out} \cup V_{loc} \cup V_{bar} \};

// perform delayed actions of previous macro step
for (y, v) \in D prv do
    ε new = [ε new]_y
end;

// start fixpoint iteration
do
    ε old = ε new;
    ℏ old = ℏ new;

    // reset the incarnation levels
    for y \in V_{loc} do
        ℏ old = [ℏ old]_y;
    end;

    // reset the barriers
    for λ \in V_{bar} do
        ε old = [ε old]^{ε prv (λ)};
    end;

    // elaboration phase
    (ε new, ℏ new, ℋ new, D new, C new) = ExecWithoutPmpt(ε old, ℏ old, ℋ prv, T, J);

// react to absence of actions on event variables
for y \in V_{event} \cup V_{event} do
    if (y, 0) \notin C new then ε new = SetEnv(ε new, y, 0, false);
end;

// react to absence of actions on state variables
for y \in V_{state} \cup V_{state} do
    if (y, 0) \notin C new then ε new = SetEnv(ε new, y, 0, GetEnv(ε prv, y, 0));
end;

// react to absence of actions on incarnated local variables
for y \in V_{event} \cup V_{state} do
    for i = 1 \ldots ℏ new(y) do
        if (y, i) \notin C new then SetEnv(ε new, y, i, default(y));
    end
end

while (ε old \neq ε new);

// transfer value of highest reincarnated value to the depth
for y \in V_{event} \cup V_{state} do
    SetEnv(ε new, y, 0, GetEnv(ε new, y, ℏ new(y)));
end

return (ε new, ℋ new, D new);
end

Fig. 6.19. Executing a Macrostep for Jobs within a Preemption Tree
Synthesis of Asynchronous Systems

Dataflow process networks (DPNs) [60, 159, 164, 198] provide a quite simple, but very powerful model of computation. DPNs consist of a fixed and finite number of processes that are statically connected with other processes of the DPN via FIFO buffers. Each FIFO buffer has a unique source and a unique sink process that can either write data values to or read data values from the FIFO buffer. The size of the FIFO buffer is unlimited in the general model of computation, but has to be finite for practical implementations which raises the question of \textit{boundedness}, i.e., whether a DPN can be run with buffers of finite size.

Besides the communication architecture, the main question for a model of computation is how and when computations are triggered. In a DPN, there is no global coordination of the computation of the single processes. Instead, the single processes can perform their computations independent of the other processes and can start as soon as the data values required for a computation step are available. For this reason, one often says that the processes are \textit{triggered by the arrival of input values}. Another point of view may be that the processes are triggered by the need to compute their output values, which leads to an \textit{output driven (or demand driven) computation} [103, 124, 205, 206, 254]. Yet another point of view can be taken, if one assumes that the single dataflow processes can be executed by an additional enable signal that has to know that input values are available. These enable signals may therefore be viewed as clocks so that a \textit{clock-driven computation} would be obtained. In particular, if a static schedule can be computed in advance, one can implement that schedule by firing the different nodes at the corresponding points of time. Synchronous, single-rate, cyclo-static, and multi-rate dataflow process networks can be efficiently implemented in this way (see Section 7.2). Hence, DPNs offer therefore many different ways for coordinating their execution. Clearly, we wish that the results of their computations should be independent of the particular way they were obtained. As we will see, this is not the case for arbitrary DPNs, so that we have to impose certain restrictions that guarantee the determinacy of a DPN.
Before determining the global coordination of the computations of a DPN, the behavior of its process nodes has to be determined, which is typically done by means of firing rules. These rules specify how many data values are read from the different input buffers of a node, and which and how many output values are generated on its output buffers as a result of the execution step. Such firing rules are also used in Petri nets or marked graphs [4, 74, 181, 203, 204, 216] that therefore share some similarities with DPNs [142]. The difference is, however, that DPNs have a specialized communication due to the point-to-point communication via FIFO buffers. DPNs resemble therefore more to marked graphs [71, 135, 181], but are more general than these, since DPNs can produce and consume more than one data values when a node fires.

DPNs are very popular, perhaps for the following obvious advantages: First, the implementation of a DPN on a distributed network of computer systems is fairly simple, since all one has to take care of is that the communication over the FIFO buffers is thereby established. This is the result of the very robust communication that most often leads to a latency-insensitive behavior of the network. Second, in contrast to the synchronous model of computation or discrete-event driven languages, DPNs do not demand the use of special programming languages. Instead, even traditional sequential programming languages may be used for the implementation of a node's behavior.

While the programming languages used to implement the behavior of the nodes may be freely chosen, the behavior of a node must fulfill some minimal requirements in order to ensure the determinacy of a DPN, which means that the DPN implements a function that maps input streams to output streams. For this reason, one has to carefully describe the semantics of DPNs. To this end, typically operational and denotational semantics are considered: The operational semantics determines the single execution steps of a DPN based on the firing rules of its nodes, and the denotational semantics is based on the composition of functions that are associated with the nodes of a DPN. As observed by Kahn [138, 139], the meaning of feedback loops can be naturally described by least fixpoints in the denotational semantics. These fixpoints exist at least if the functions implemented by the basic nodes are continuous functions which motivates the restriction to continuous base functions.

Clearly, it is desirable that both semantics should coincide to make sure that a robust semantics has been found that is amenable to abstractions made by the denotational semantics. However, it is known from functional programming languages that this is not trivial [39, 56, 178, 188, 189, 208]: Plotkin [208] studied the relationship of operational and denotational semantics for PCF (Programs for Computable Functions), a programming language he derived from the λ-calculus in analogy to LCF (the Logic of Computable Functions) [112, 236]. He observed that the operational semantics implies the denotational one, but the converse is not true if continuous functions are used in the denotational semantics (since there are continuous functions that can not be implemented by sequential programs). For this reason, Milner [178] posed the full abstraction problem [56, 189] that consists of determining a
suitable subset of the continuous functions such that the operational and the denotational semantics coincide. To this end, he and independently Vuillemin [258], as well as others, defined sets of *sequential functions* as a potential suitable subset. As Plotkin already proved in [208] that the addition of a parallel if-then-else operator (see also [177]) solves the full abstraction problem, the notion of sequentiality is best explained by this operator: In functional languages, it is possible that the evaluation of a function fails for some arguments, e.g. since invalid arguments were given (like division by zero) or since the evaluation does not terminate. In these cases, the result value \( \perp \) is used in the semantics. The sequential and parallel if-then-else operators are then described as follows:

\[
\text{ITE}_s(\alpha, \beta, \gamma) := \begin{cases} 
\beta & \text{if } \alpha = 1 \\
\gamma & \text{if } \alpha = 0 \\
\perp & \text{if } \alpha = \perp
\end{cases}
\]

\[
\text{ITE}_p(\alpha, \beta, \gamma) := \begin{cases} 
\beta & \text{if } \alpha = 1 \\
\gamma & \text{if } \alpha = 0 \\
\beta & \text{if } \alpha = \perp \text{ and } \beta = \gamma \\
\perp & \text{if } \alpha = \perp \text{ and } \beta \neq \gamma
\end{cases}
\]

It is easily seen that both operators are equally expressive (note that the result of the comparison \( \beta = \gamma \) is either 1 or 0):

- \( \text{ITE}_s(\alpha, \beta, \gamma) = \text{ITE}_p(\alpha, \text{ITE}_p(\alpha, \beta, \perp), \text{ITE}_p(\alpha, \perp, \gamma)) \)
- \( \text{ITE}_p(\alpha, \beta, \gamma) = \text{ITE}_s(\beta = \gamma, \beta, \text{ITE}_s(\alpha, \beta, \gamma)) \)

In general, sequentiality means that there is a sequential order in which the arguments of a function are evaluated to determine the function’s value. For example, in \( \text{ITE}_s \), we first have to evaluate the condition \( \alpha \), and depending on its result, we either evaluate \( \beta \) or \( \gamma \). In the parallel version \( \text{ITE}_p \), all arguments are evaluated in parallel, and the function’s value is determined as soon as enough information is given by these evaluations.

Since sequential functions have not solved the full abstraction problem, Berry introduced the set of *stable functions* [39, 44, 56, 188, 189]. Stable functions form a subset of continuous functions, but are a superset of the sequential functions. Berry proved the strictness of these set inclusions by means of his Gustave function (see page 237) and was able present a solution to the full abstraction problem.

We will see that using continuous functions for the operational semantics is, similar to functional programming languages, also too generous for DPNs. For this reason, we have to impose further restrictions to the base functions so that the operational and denotational semantics of DPNs coincide. To explore this, we describe the *operational semantics of DPNs in terms of rewrite systems* so that their determinacy becomes equivalent to the *confluence* of different execution steps. Similar to the situation for functional programming languages, we will see that there are certain functions that cannot be implemented without restricting the schedule of the execution of the DPN’s nodes. We are therefore interested in the subset of continuous functions whose operational semantics lead to confluent rewrite systems. It is not too surprising that the observations made for DPNs are analogous to the results obtained for
functional programming languages, since one may also view the execution of DPNs as the evaluation of a functional expression. For this reason, we will see that the subset of confluent continuous functions are exactly Berry’s stable functions (see Section 7.1.5).

Besides the full abstraction problem of the operational and denotational semantics, there are further problems concerning the semantics of DPNs: Further problems are given by modularization where one wants to collapse a set of process nodes into a single node. However, while this seems to be simple, it is not at all simple to implement the firing rules of the thereby generated nodes [164, 172, 199, 255]. Moreover, hiding the local nodes and FIFO buffers may appear as a nondeterministic behavior of the entire network, since (1) the points of time when different outputs appear is not determined by the contained DPN and (2) the behavior now depends on the contents of the local FIFO buffers, so that the entire behavior now depends on a local state. For this reason, the description of DPNs with nondeterministic nodes was for some time a hot-topic in research [23, 77, 99, 144, 147, 192, 241–243] where anomalies like the Keller anomaly [143] or the Brock-Ackermann anomaly [55] demonstrated that simple solutions like describing the behavior of a node by relations instead of functions does not work. In the meantime, different solutions have been found that are based on trace-based semantics [23, 147], game-based semantics [99] or clever use of functionals [243].

Finally, once the problems concerning determinacy and modularity are solved, further problems have to be addressed for practical implementations of DPNs. The most difficult problem is the boundedness problem where one has to check whether a given DPN can be run with FIFO buffers of finite size. If so, one would moreover be interested in the minimal sizes as well as in a schedule to fire the nodes so that these sizes are sufficient. For general DPNs, this problem is undecidable so that implementations with finite memory, as required by hardware circuits, are in general not possible. For this reason, some restrictions of DPNs have been considered that have decidable boundedness problems. In particular, the synchronous dataflow (SDF) of Lee and Messerschmitt [159, 162, 163] is such a decidable class. In these DPNs, the number of values consumed from the input buffers and produced at the output buffers may be different for each buffer, but must be the same for all possible execution steps of each node. The notion ‘synchronous’ is probably misleading, since it is not necessarily the case that all nodes fire with a single clock. However, it is the case that the computation can be split into periods where a static schedule is repeated in each period to fire the nodes. The restriction where all nodes fire synchronously are sometimes called single-rate DPNs, while synchronous dataflow networks are then called multi-rate DPNs [49]. A generalization of these are the cyclo-static DPNs of Lauwereins [49, 96, 97, 260], where each node has a regular firing pattern so that the entire network can still have a static schedule that guarantees its boundedness. On the other hand, even adding an if-then-else node that then defines Boolean DPNs [58, 59] leads to an undecidable boundedness problem (but to Turing-completeness).
Since the behavior of the single nodes are almost arbitrary, the granularity of the node’s behaviors compared to the entire dataflow network can also be freely chosen. Therefore, dataflow process networks have been used in many areas and they are still in use. They have been even proposed for the construction of computer architectures [10–15, 17, 50, 72, 76, 79–83, 83, 84, 105, 113, 114, 158, 193, 194, 221, 238, 239, 257, 266] as well as to specialized dataflow programming languages [2, 136, 261] including Lucid [18, 259], VAL [3], SISAL [62, 63], Id [14, 184], the synchronous language Lustre [118–120], and the polychronous language SIGNAL [36, 37, 107, 156, 157]. Dataflow programming is one of the answers to Backus’s question on whether programming can be liberated from the von Neumann style [24].

In this chapter, we introduce the formal foundation of the semantics of dataflow process networks. To this end, we refer the reader to Section A.3 for the mathematical properties of streams where it is shown that they form a complete partial order as well as a complete metric space. This is very important since we can make use of theorems to guarantee the existence of fixpoints that are used in the denotational semantics. We then introduce the syntax and semantics of DPNs in Section 7.1, where we first describe DPNs in terms of rewrite systems in Section 7.1.3 so that we can define their operational semantics in Section 7.1.4. The denotational semantics that is based on the mathematical properties of the streams is given in Section 7.1.2, and their equivalence is discussed in Section 7.1.2. Finally, we discuss restricted DPNs that have decidable boundedness problems and show how static schedules can be derived for them in Section 7.2.

7.1 Dataflow Process Networks

In this section, we introduce syntax and semantics of dataflow process networks. In particular, we will define an operational and a denotational semantics and will discuss the equivalence of these semantics which leads to the full abstraction problem that has been posed by Plotkin and Milner for functional languages [39, 178, 188, 189, 208]. The operational semantics will be given in terms of rewrite systems, and the denotational semantics is based on the Kahn principle [138, 139] where the semantics of feedback edges can be described as a least fixpoint in the denotational semantics.

7.1.1 Syntax of DPNs

Although the graphical representation of dataflow networks is commonly considered as a big advantage of this model of computation, we will introduce a textual representation. The reason for this is simply that we are then able to precisely specify a language via its abstract syntax and can formally reason about its properties.
Analogous to the previous section that presented the mathematical theory of streams, we will not distinguish data types for different streams. Instead, we assume that all the data values that are used as inputs and outputs of functions belong to a set $\mathcal{D}$ that is not further specified. It may be viewed as the union of the domains of all used data types. Of course our ‘programming language’ has to make use of expressions of type $\mathcal{D}$ as well as of expression of type $\mathcal{D}^\infty$, and it will contain processes of type $\text{Stream}_m^D \rightarrow \text{Stream}_n^D$. To this end, we first introduce the set of expressions. To this end, we use a notation that is related to the description of functional languages like ML [123, 190]. In particular, we will make use of pattern matching for streams: For our domain $\mathcal{D}$, we assume to have some variables $V_D$ of type $\mathcal{D}$ (i.e., representing elements of $\mathcal{D}$) as well as some constants $c \in \mathcal{D}$. To describe stream expressions, we use the constant $[\ ]$ to denote the empty stream, and $(e :: \sigma)$ to denote a stream that is obtained by adding a new prefix consisting of an expression $e$ of type $\mathcal{D}$ and a stream expression $\sigma$.

**Definition 7.1 (Expressions, Substitutions, Matchings, and Unifiers).**

Given a set of constants $\mathcal{D}$ and a set of variables $V_D$ of type $\mathcal{D}$, the expressions $\text{Expr}_D$ of type $\mathcal{D}$ are defined as $\text{Expr}_D := \mathcal{D} \cup V_D$. Given a further set of variables $V_D^\infty$ of type $\mathcal{D}^\infty$, the following are stream expressions $\text{Expr}_D^\infty$:

- $[\ ]$ is in $\text{Expr}_D^\infty$
- every $L \in V_D^\infty$ is in $\text{Expr}_D^\infty$
- if $e \in \text{Expr}_D$ and $\sigma \in \text{Expr}_D^\infty$, then $(e :: \sigma)$ is in $\text{Expr}_D^\infty$

A substitution is a function $\varrho : V_D \cup V_D^\infty \rightarrow \text{Expr}_D \cup \text{Expr}_D^\infty$ so that $x \in V_D$ is mapped to $\varrho(x) \in \text{Expr}_D$ and $L \in V_D^\infty$ is mapped to $\varrho(L) \in \text{Expr}_D^\infty$. Substitutions are extended from variables to expressions $\text{Expr}_D^\infty$ as follows:

- $\varrho([\ ]):= [\ ]$
- $\varrho(L)$ is the value of $L$ under $\varrho$
- $\varrho((e :: \sigma)) := (\varrho(e) :: \varrho(\sigma))$

An expression $\sigma_1$ matches to another expression $\sigma_2$ if there is a substitution $\varrho$ such that $\varrho(\sigma_1) = \sigma_2$. Finally, if $\varrho(\sigma_1) = \varrho(\sigma_2)$ holds, then $\varrho$ is a unifier of $\sigma_1$ and $\sigma_2$.

In order to simplify the notation, we alternatively write $[1, 2, 3]$ instead of $(1 :: (2 :: (3 :: [\ ])))$ to denote a stream consisting of the integers 1, 2, and 3 in that order.

A pattern is then simple a stream expression that is used to match with a concrete stream of values. For example, $[1, x, 3]$ matches with each stream of integers that contains three elements where the first is 1 and the third is 3, while the second one may be any integer. While the patterns $[\ ]$ and $[e_1, \ldots, e_n]$ denote streams of a fixed finite length, the pattern $L$ matches any stream and $(1 :: (2 :: L))$ any stream of integers with prefix $[1, 2]$. 
A basic function \( f_i : (\mathcal{D}^\ast)^m \to (\mathcal{D}^\ast)^n \) used in the definition of a DPN as given in Definition 7.2 is then described by a table that lists different patterns \( P_{i,j} \in \text{Expr}_{\mathcal{D}^\infty} \) and corresponding result expressions \( \sigma_{i,j} \in \text{Expr}_{\mathcal{D}^\infty} \) with the restriction that the results \( \sigma_{i,j} \) may only consist of the variables \( x \in \mathcal{V}_D \) that occur in the patterns of the same row (note that they must not contain variables \( L \in \mathcal{V}_{\mathcal{D}^\infty} \) used in the corresponding patterns):

\[
\begin{array}{c|c|c|c|c|c|c|c}
& x_1 & \cdots & x_m & y_1 & \cdots & y_n \\
P_{1,1} & \cdots & P_{1,m} & \sigma_{1,1} & \cdots & \sigma_{1,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
P_{p,1} & \cdots & P_{p,m} & \sigma_{p,1} & \cdots & \sigma_{p,n} \\
\end{array}
\]

Each row of such a table describes a possible firing rule, i.e., a computation step of the specified function that is to be read as follows: If a pattern \( P_{i,j}, \ldots, P_{i,m} \) matches the input streams \( x_1, \ldots, x_m \), then the corresponding expressions of type \( \mathcal{D} \) that occur in the patterns are consumed from the input streams and are used to compute the outputs by evaluating the expressions \( \sigma_{1,1}, \ldots, \sigma_{1,n} \) listed on the result columns of the table. These outputs are then added to the right of the so far computed output streams. To be precise, the consumption of values from the input streams is defined as follows (see also Definition 7.4):

- \([\ ]\) consumes no values from input streams
- \( L \in \mathcal{V}_{\mathcal{D}^\infty} \) consumes no values from input streams
- \((e :: \sigma) \in \text{Expr}_{\mathcal{D}^\infty}\) consumes the leading value of an input stream, and all values consumed by \( \sigma \) of the remaining stream

As can be seen, always a prefix of the input stream is consumed by a firing rule. For this reason, the computation for given input streams is performed by consuming prefixes of these input streams step by step and by producing the output streams by concatenating suffixes in these steps to the so far generated finite output prefixes. While this already explains the intuitive meaning of base functions, their precise semantics are given in the following sections.

The firing rules of a node may also be split into an initial and remaining behavior that are separated by a horizontal line between the different firing rules. Hence, the general scheme of such a node looks as follows, where at the initial point of time, the firing rules with the patterns \( I_{i,j} \) are tried, while after the initial point of time only the firing rules with the patterns \( P_{i,j} \) are tried.
We can now define the composition of single base functions. For base functions of type \(\text{Stream}^1 \rightarrow \text{Stream}^1\), this would be simply the composition of these functions, but for the general case \(\text{Stream}^m \rightarrow \text{Stream}^n\), this is not so simple, and requires the use of local variables. For this reason, we connect the single processes that execute base functions via buffers so that each buffer has exactly one producer process that writes data values to the buffer and also exactly one consumer process that reads data values from the buffer. Thus, we obtain the following formal definition of a DPN:

**Definition 7.2 (Dataflow Process Networks (DPNs)).** Given sets of input variables \(V_{\text{in}} = \{x_1, \ldots, x_m\}\), output variables \(V_{\text{out}} = \{y_1, \ldots, y_n\}\), and local variables \(V_{\text{loc}} = \{z_1, \ldots, z_k\}\) as well as a set of basic function symbols \(F\) that are specified by firing rules, an equation system of the following form is a DPN:

\[
\begin{align*}
(u_{1,1}, \ldots, u_{1,q_1}) &= f_1(v_{1,1}, \ldots, v_{1,p_1}) \\
& \vdots \\
(u_{\ell,1}, \ldots, u_{\ell,q_\ell}) &= f_\ell(v_{\ell,1}, \ldots, v_{\ell,p_\ell})
\end{align*}
\]

Thereby, \(f_i \in F\) are base functions, \(v_{i,j} \in V_{\text{in}} \cup V_{\text{loc}}\), \(u_{i,j} \in V_{\text{out}} \cup V_{\text{loc}}\) such that the number of variables in each equation must be consistent with the type of the corresponding base function \(f_i\). Moreover, each input variable has exactly one occurrence in one of the right hand sides, each output variable has exactly one occurrence on the left hand sides, and each local variable has exactly one occurrence in one of the right hand sides and exactly one occurrence on the left hand sides.

A DPN can therefore read values from the variables \(V_{\text{in}} \cup V_{\text{loc}}\), and can write values to the variables \(V_{\text{loc}} \cup V_{\text{out}}\), as will be described in detail by the operational semantics in the next section. The local variables \(V_{\text{loc}}\) are used to store an internal state of the DPN so that DPNs have an internal memory of infinite size.

As an example, consider the DPN shown in Figure 7.1. It consists of three process nodes that perform the functions even, odd, and merge. There are no inputs, but one output stream \(y\), and there are four local variables \(z_1, z_2, z_3,\) and \(z_4\). The behavior of the nodes even, odd, and merge are given on the right of Figure 7.1, where the behaviours of even and odd distinguish between their initial and remaining behaviors: Initially, even does not read values, but writes
the number 0 both to \( z_1 \) and \( z_3 \), and later on, it reads one value \( i \) of \( z_1 \) and writes the number \( i + 2 \) both to \( z_1 \) and \( z_3 \). The node odd works similarly with the only difference that it initially writes 1 instead of 0 to its buffers. The function merge tries to read values of its input buffers \( z_3 \) and \( z_4 \) and writes these to the output buffer \( y \) in an ordering where the value read from \( z_3 \) occurs first. If there should be no value in one of the input buffers, the node proceeds with copying the value of the other buffer to the output buffer \( y \).

We will see in the following sections that this DPN has some problems that do not allow us to define an operational semantics. Also the denotational semantics runs into problems, as we will see below.

### 7.1.2 Denotational Semantics of DPNs

In this section, we will define a denotational semantics for DPN that associates with each \( N \in \text{DPN} \) a function of type \( \text{Stream}^m_D \rightarrow \text{Stream}^n_D \), where \( m := |\mathcal{V}_{\text{in}}| \) and \( n := |\mathcal{V}_{\text{out}}| \). To this end, we start with defining a denotational semantics for the expressions, the for the base functions, and finally for entire DPNs.

In general, a denotational semantics [180] as developed by Scott and Strachey gives certain syntactic objects a meaning by mapping them to objects that already have a meaning. For this reason, a denotational semantics consists of the definition of the corresponding maps from syntactic objects to appropriate elements. In case of expressions, we obviously choose the elements of our
domain of values $\mathcal{D}$ for the data values, and the streams $\mathcal{D}^\infty$ for the stream expressions. The denotational semantics of based on variable assignments, while the constants already essentially interpret themselves:

**Definition 7.3 (Denotational Semantics of Expressions).** Given a variable assignment $\xi : \mathcal{V}_D \cup \mathcal{V}_D^\infty \rightarrow \mathcal{D} \cup \mathcal{D}^\infty$ such that $\xi(x) \in \mathcal{D}$ for $x \in \mathcal{V}_D$ and $\xi(L) \in \mathcal{D}^\infty$ for $L \in \mathcal{V}_D^\infty$, we define the semantics of expressions as follows:

- $\{x\}_\xi := \xi(x)$ for $x \in \mathcal{V}_D$
- $\{c\}_\xi := c$ for $c \in \mathcal{D}$
- $\{L\}_\xi := \xi(L)$ for $L \in \mathcal{V}_D^\infty$
- $\{[]\}_\xi := []$
- $\{(e :: \sigma)\}_\xi := ([e]_\xi :: [\sigma]_\xi)$

We should remark here that we use the same constants and symbols at the meta level, e.g., we use $[]$ both as expression and as its meaning for the empty stream. The function symbols $f$ for the base functions must be mapped by an interpretation function $I$ to a function of type $\text{Stream}^m_\mathcal{D} \rightarrow \text{Stream}^n_\mathcal{D}$ so that the firing rules given for $f$ are respected. To this end, we first define the suffix that is left after a pattern matching in case a rule is used to fire a function $f$:

**Definition 7.4 (Suffix of Patterns).** We define the function $\text{Suffix} : \text{Expr}_\mathcal{D}^\infty \rightarrow \mathcal{V}_D^\infty$ as follows:

- $\text{Suffix}([]) := []$
- $\text{Suffix}(L) := L$
- $\text{Suffix}(e :: \sigma) := \text{Suffix}(\sigma)$

According to the consumption of values, the function $\text{Suffix}$ computes the suffix that is left in the corresponding stream after $f$ has been fired. Note that for every pattern $P$, $\text{Suffix}(P)$ is either a variable of type $\mathcal{D}^\infty$ or the empty stream constant $[]$. Using the function $\text{Suffix}$, we make the following definition:

**Definition 7.5 (Pattern-Consistent Interpretation).** Assume that a base function $f$ has a firing rule with patterns $(P_1, \ldots, P_m)$ for which output sequences $(\sigma_1, \ldots, \sigma_n)$ have to be produced. An interpretation $I$ of the base function $f$ is called pattern-consistent iff we have for every variable assignment $\xi : \mathcal{V}_D \cup \mathcal{V}_D^\infty \rightarrow \mathcal{D} \cup \mathcal{D}^\infty$ the following equation

$$I(f) \left([P_1]_\xi, \ldots, [P_m]_\xi\right) = \left([\sigma_1]_\xi \cdot \beta_1, \ldots, [\sigma_n]_\xi \cdot \beta_n\right)$$

where $(\beta_1, \ldots, \beta_n) := I(f) \left([\text{Suffix}(P_1)]_\xi, \ldots, [\text{Suffix}(P_m)]_\xi\right)$.

Clearly, we should only consider pattern-consistent interpretations for the denotational semantics of DPNs, since otherwise the base functions do not have the intended meaning. We can now define the denotational semantics as follows:
Definition 7.6 (Behaviors of Dataflow Process Networks (DPNs)). For every DPN \( N \) as described in Definition 7.2 and every interpretation \( I \) of the function symbols \( f_i \), the set of behaviors \( \text{Bhv}_I(N) \) of \( I \) and \( N \) is defined as the set of variable assignment \( \xi \) satisfying the following equations:

\[
\begin{cases}
(\{u_{1,1}\}_\xi, \ldots, \{u_{1,q_1}\}_\xi) = I(f_1)(\{v_{1,1}\}_\xi, \ldots, \{v_{1,p_1}\}_\xi) \\
\vdots \\
(\{u_{\ell,1}\}_\xi, \ldots, \{u_{\ell,q_\ell}\}_\xi) = I(f_\ell)(\{v_{\ell,1}\}_\xi, \ldots, \{v_{\ell,p_\ell}\}_\xi)
\end{cases}
\]

Definition 7.7 (Denotational Semantics). Given a DPN \( N \) as described in Definition 7.2, we first split the equations on tuples into single equations by decomposing the base functions \( h_i : \text{Stream}^m_D \rightarrow \text{Stream}^n_D \) into component functions of type \( \text{Stream}^{m+k}_D \):

\[
\begin{align*}
y_1 &= f_1(x_1, \ldots, x_m, z_1, \ldots, z_k) \\
\vdots \\
y_n &= f_n(x_1, \ldots, x_m, z_1, \ldots, z_k) \\
z_1 &= g_1(x_1, \ldots, x_m, z_1, \ldots, z_k) \\
\vdots \\
z_k &= g_k(x_1, \ldots, x_m, z_1, \ldots, z_k)
\end{align*}
\]

Then, we define for all streams \( (u_1, \ldots, u_m) \in \text{Stream}^m_D \) and \( (v_1, \ldots, v_k) \in \text{Stream}^k_D \) a variable assignment \( \xi \) such that \( \xi(x_i) := u_i \) for all \( i \in \{1, \ldots, m\} \) and \( \xi(z_i) := v_i \) for all \( i \in \{1, \ldots, k\} \). Using \( \xi \), we define the following function \( G_{\xi} : \text{Stream}^k_D \rightarrow \text{Stream}^k_D \):

\[
G_{\xi} \begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix} := \begin{pmatrix} \{g_1(x_1, \ldots, x_m, z_1, \ldots, z_k)\}_\xi \\ \vdots \\ \{g_k(x_1, \ldots, x_m, z_1, \ldots, z_k)\}_\xi \end{pmatrix}
\]

If \( (\hat{v}_1, \ldots, \hat{v}_k) \) is the least fixpoint of the above function \( G_{\xi} \), then we define a new variable assignment \( \hat{\xi} \) such that \( \hat{\xi}(x_i) := u_i \) for all \( i \in \{1, \ldots, m\} \) and \( \hat{\xi}(z_i) := \hat{v}_i \) for all \( i \in \{1, \ldots, k\} \). Using \( \hat{\xi} \), the denotational semantics \( \llbracket N \rrbracket_x \) of \( N \) is defined as the following function \( F_{\hat{\xi}} \):

\[
F_{\hat{\xi}} \begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix} := \begin{pmatrix} \{f_1(x_1, \ldots, x_m, z_1, \ldots, z_k)\}_{\hat{\xi}} \\ \vdots \\ \{f_n(x_1, \ldots, x_m, z_1, \ldots, z_k)\}_{\hat{\xi}} \end{pmatrix}
\]

As an example, consider the DPN given in Figure 7.1. The DPN is first decomposed into scalar equations by splitting the base functions as follows:
even₁ = even₂  odd₁ = odd₂  merge

\[
\begin{array}{c|c}
  x₁ & y₁ \\
  \hline
  A & 0 \\
  (i :: A) & i + 2
\end{array}
\quad
\begin{array}{c|c}
  x₁ & y₁ \\
  \hline
  A & 1 \\
  (i :: A) & i + 2
\end{array}
\quad
\begin{array}{c|c|c|c}
  x₁ & x₂ & y \\
  \hline
  (a :: A) & (b :: B) & [a, b] \\
  (a :: A) & [b] & [a]
\end{array}
\] 

\[
\begin{array}{c|c|c|c}
  y = \text{merge}(z₃, z₄) \\
  z₁ = \text{even₁}(z₁) \\
  z₂ = \text{odd₁}(z₂) \\
  z₃ = \text{even₂}(z₁) \\
  z₄ = \text{odd₂}(z₂)
\end{array}
\]

It is easily seen that we have to determine the interpretations of the base functions even, odd, and merge such that we have the following denotations for their component functions:

1. \( I(\text{even })(\sigma) := \begin{cases} 
   [0, a₁ + 2, \ldots, aₙ + 2] & : \text{if } \sigma = [a₁, \ldots, aₙ] \\
   [0, a₁ + 2, a₂ + 2, \ldots] & : \text{if } \sigma = [a₁, a₂, \ldots] \text{ is infinite}
\end{cases} \)
2. \( I(\text{odd })(\sigma) := \begin{cases} 
   [1, a₁ + 2, \ldots, aₙ + 2] & : \text{if } \sigma = [a₁, \ldots, aₙ] \\
   [1, a₁ + 2, a₂ + 2, \ldots] & : \text{if } \sigma = [a₁, a₂, \ldots] \text{ is infinite}
\end{cases} \)
3. \( I(\text{merge})(\sigma₁, \sigma₂) := \begin{cases} 
   [a₁, b₁, \ldots, aₙ, bₙ, aₙ₊₁, \ldots, aₙ₊ₘ] & : \text{if } \sigma₁ = [a₁, \ldots, aₙ₊ₘ] \text{ and } \sigma₂ = [b₁, \ldots, bₙ] \\
   [a₁, b₁, \ldots, aₙ, bₙ, bₙ₊₁, \ldots, bₙ₊ₘ] & : \text{if } \sigma₁ = [a₁, \ldots, aₙ] \text{ and } \sigma₂ = [b₁, \ldots, bₙ₊ₘ]
\end{cases} \)

As the considered DPN has no inputs, the function \( G_\xi \) mentioned in the previous definition is defined as follows:

\[
G_\xi \begin{pmatrix} v₁ \\ v₂ \end{pmatrix} := \begin{pmatrix} I(\text{even })(v₁) \\ I(\text{odd })(v₂) \end{pmatrix}
\]

We can now compute the least fixpoint of \( G_\xi \) by the usual fixpoint iteration. We thereby obtain the following values:

\[
G^n_\xi \begin{pmatrix} [0, 2, \ldots, 2n - 2] \\ [1, 3, \ldots, 2n - 1] \\ [0, 2, \ldots, 2n - 2] \\ [1, 3, \ldots, 2n - 1] \end{pmatrix} = \begin{pmatrix} [0, 2, 4, \ldots] \\ [1, 3, 5, \ldots] \\ [0, 2, 4, \ldots] \\ [1, 3, 5, \ldots] \end{pmatrix}
\]

As \( G^n_\xi \) is not monotone. The assignment \( \hat{\xi} \) is therefore defined such that \( \hat{\xi}(z₁) = \hat{\xi}(z₃) = [0, 2, 4, \ldots] \) contains the even numbers and \( \hat{\xi}(z₂) = \hat{\xi}(z₄) = [1, 3, 5, \ldots] \) contains the odd numbers. For this reason, it is now easy to see that the denotational semantics of the entire DPN is the following function (since we have no input streams, it is actually just the stream of all numbers):

\[
F_\hat{\xi}() = I(\text{merge})(z₃, z₄) = [0, 1, 2, 3, 4, 5, \ldots]
\]

We will however see in the following section that the operational semantics of this DPN is not well-defined since the function \( I(\text{merge}) \) is not monotone. Note that this does not yet make trouble for the denotational semantics, since
the fixpoint is only computed for the functions \( I(\text{even}) \) and \( I(\text{odd}) \). However, even for these simple functions, one may run into trouble if one wants to use the additional wrong implementations:

<table>
<thead>
<tr>
<th>even</th>
<th>odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z_1 )</td>
<td>( z_1' )</td>
</tr>
<tr>
<td>( { } )</td>
<td>( {0} )</td>
</tr>
</tbody>
</table>

\( i :: A \)

The above implementations do not distinguish between an initial and remaining behavior. The idea is that the buffers \( z_i \) are initially all empty, so that only the matching rules in the first lines can be applied at that point of time. Later on, these node will read the only value that is contained in the buffers \( z_i \) and will increase it by 2. While this works in an operational semantics, it does not work in the denotational semantics. The trouble already starts with determining pattern-consistent denotations. The only pattern-consistent denotations are the following functions:

- \( I(\text{even})_i(\sigma) := \begin{cases} \{a_1 + 2, \ldots, a_n + 2, 0\} & : \text{if } \sigma = \{a_1, \ldots, a_n\} \\ \{a_1 + 2, a_2 + 2, \ldots\} & : \text{if } \sigma = \{a_1, a_2, \ldots\} \text{ is infinite} \end{cases} \)
- \( I(\text{odd})_i(\sigma) := \begin{cases} \{a_1 + 2, \ldots, a_n + 2, 1\} & : \text{if } \sigma = \{a_1, \ldots, a_n\} \\ \{a_1 + 2, a_2 + 2, \ldots\} & : \text{if } \sigma = \{a_1, a_2, \ldots\} \text{ is infinite} \end{cases} \)

Unfortunately, these functions are not monotone, and they do not have fixpoints, so that we cannot determine a denotational semantics for this DPN. The fixpoint iteration would generate the following non-convergent sequence of values:

\[
G_n^\xi(\begin{pmatrix} \{\} \\ \{\} \\ \{\} \end{pmatrix}) = \begin{pmatrix} \{2n - 2, 2n - 4, \ldots, 2, 0\} \\ \{2n - 1, 2n - 3, \ldots, 3, 1\} \\ \{2n - 2, 2n - 4, \ldots, 2, 0\} \\ \{2n - 1, 2n - 3, \ldots, 3, 1\} \end{pmatrix}
\]

The problem is that while in the operational semantics, the matching with \( \{\} \) is only done at the initial point of time, the denotational semantics determines a function \( I(\text{even}) \) where \( \{\} \) is only matched once all elements of a finite stream have been consumed by the other firing rule. Thus, instead of applying the first firing at the initial point of time, it is erroneously used after consumption of a finite sequence which was not the intention.

### 7.1.3 DPNs as Rewrite Systems

To describe the operational semantics of a DPN, we must first determine the semantics of the basic functions. To this end, recall that each firing rule therefore consumes values from the input streams corresponding to the above rules after matching with the used patterns. The computation of the output values according to the firing rule will then write these values to the corresponding output streams. For this reason, each function may be viewed as a read/write
head that works on a given input stream and translates it to a corresponding output stream while reading it.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y_1$</th>
<th>$y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1 :: L_1) (a_1 :: L_1)$</td>
<td>$(b_1 :: (b_2 :: L_2))$</td>
<td>$[a_1, b_2]$</td>
<td>$[b_1, b_1]$</td>
<td></td>
</tr>
<tr>
<td>$(0 :: L_1) (a_1 :: (a_2 :: L_1)) (b_1 :: L_2)$</td>
<td>$[b_1, a_2]$</td>
<td>$[a_1]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$
f = \begin{pmatrix} 1 & 0 & 1 \\ a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} \cdot f = \begin{pmatrix} 1 & 0 & 1 \\ a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} \cdot f = \begin{pmatrix} 0 & 1 \\ a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\ b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} \cdot f = \begin{pmatrix} 1 \end{pmatrix} \cdot f = \begin{pmatrix} a_4 & a_5 & a_6 & a_7 \\ b_5 & b_6 & b_7 \end{pmatrix} \cdot f = \begin{pmatrix} a_5 & a_6 & a_7 \\ b_5 & b_6 & b_7 \end{pmatrix}
$$

Fig. 7.2. Example computation of a DPN

For example, consider the function table on top of Figure 7.2 and the example computation below that table. This function can perform computations in two ways specified by its two firing rules given in its function table: The first firing rule requires that the first input stream $x_1$ starts with 1, then one value $a_1$ must be taken from $x_2$ and two values $b_1, b_2$ must be taken from $x_3$ to generate the output sequence $([a_1, b_2], [b_1, b_1])$ that is added to the right of the so far computed output stream. The second firing rule requires that the first input stream $x_1$ starts with 0, then two values $a_1, a_2$ must be taken from $x_2$ and one value $b_1$ must be taken from $x_3$ to generate the output sequence $([b_1, a_2], [a_1])$ that is added to the right of the so far computed output stream.

Hence, a computation of a DPN is performed as shown in the lower part of Figure 7.2. Note that even though some of the components of the input stream still have values at the end, the computation terminates if there is no further possibility to fire the DPN once more. As can be seen, the computations of a DPN are very similar to the transformations performed by a term rewrite system [21, 22, 86–88, 128, 137, 145, 207, 252], so that a lot of knowledge is shared with rewrite systems. We will therefore consider some basic definitions and results of rewrite systems in the following for understanding the
7.1 Dataflow Process Networks

computations of DPNs. To this end, we define a reduction relation \( \leadsto_N \) for every DPN \( N \):

**Definition 7.8 (State Transition System of DPNs).** Given a DPN \( N \) with input variables \( V_{in} \), output variables \( V_{out} \), local variables \( V_{loc} \), and a set of basic function symbols \( F \). The state transition system \( T_N = (S_N, I_N, \leadsto_N) \) of \( N \) consists of states \( S_N \) that are variable environments \( E : V_{in} \cup V_{out} \cup V_{loc} \rightarrow D^\infty \) that map each variable of \( V_{in} \cup V_{out} \cup V_{loc} \) to a stream \( D^\infty \). A state transition \( E \leadsto_N E' \) is obtained by firing a set of nodes/equations and modifying the variables’ values in \( E \) according to the consumption and production of values specified by the used firing rules. Initial states \( I \subseteq S_N \) are those states \( E \in S_N \), where \( E(x) = [] \) holds for all \( x \in V_{out} \cup V_{loc} \). A state \( E \in S_N \) without transition \( E \leadsto E' \) is called a final state.

The state transition system \( T_N \) covers all possible ways how computations of the DPN \( N \) can be performed. For example, a DPN \( N \) with the single node \((y_1, y_2) = f(x_1, x_2, x_3)\) using the function \( f \) specified in Figure 7.2 would generate a state transition system \( T_N \) that contains the path shown in Figure 7.3. In that figure and also in the following, we describe the variable environments \( E \) by listing their mapping for every variable of the DPN.

As can be seen, the path shown in Figure 7.3 is the only computation that can be performed the DPN \((y_1, y_2) = f(x_1, x_2, x_3)\). In general, it can be possible that different choices are possible, so that the computation branches into several successor states. There are two reasons for such branchings:

1. A single node may have different choices to fire for its current input streams, i.e., more than one of its firing rules are enabled.
2. More than one node can fire, so that a subset of the enabled nodes is selected for firing. In this case, a successor \( E' \) is obtained for every possible nonempty subset of the enabled nodes.

While different ways to fire one node is typically not desired at a first glance, the second reason that allows subsets of enabled nodes to fire, is wanted since this gives DPNs the freedom to execute asynchronously. However, if one wants to collapse DPNs into single nodes to introduce modularity of DPNs, the second reason will lead to the first one also. Hence, we have to accept both reasons if modularity should be possible.

If a subset of enabled nodes is selected for firing, it is desirable that those nodes that were enabled, but not fired, remain enabled also in the next state. However, this is not always the case, since it can happen that a node can fire with a firing rule that matches patterns of the form \([[]]\) or \([e_1, \ldots, e_n]\), i.e., that makes use of patterns that do not only match to a prefix of the input stream, but match to a finite stream. If the producer of that input stream will fire and will thereby add a sequence to that input stream, the consumer node that was enabled before will no longer be enabled. For this reason, the so-called diamond property is in general not given.
Hence, if the DPN should compute uniquely determined output streams for given input streams, we have to restrict the DPNs so that branchings that are possible do not lead to different results for the output streams at the end. Clearly, we can define a uniquely determined output value for a finite input stream if all possible paths starting in initial states finally meet in the same final state $E_f$. This property is well-known in rewrite systems, and is commonly called confluence that may be defined in various ways (see e.g. [127]). We therefore study different notions of confluence in the following to determine what is required for DPNs.
Definition 7.9 (Confluence Properties). For any binary relation \( \rightsquigarrow \) on a set \( \mathcal{E} \), we define the following derived relations:

- \( x \rightsquigarrow^i y \) is the \( i \)-th product of \( \rightsquigarrow \), i.e.,
  - \( x \rightsquigarrow^0 y :\iff x = y \)
  - \( x \rightsquigarrow^{i+1} y :\iff x \rightsquigarrow^i y \lor \exists z. \ x \rightsquigarrow^i z \land z \rightsquigarrow y \)
- \( x \rightsquigarrow^+ y :\iff \exists i > 0. \ x \rightsquigarrow^i y \)
- \( x \rightsquigarrow^* y :\iff \exists i. \ x \rightsquigarrow^i y \)
- \( x \rightsquigarrow^* y :\iff x = y \lor x \rightsquigarrow y \)
- \( x \rightsquigarrow^* y :\iff x \rightsquigarrow y \lor y \rightsquigarrow x \)
- \( x \parallel y :\iff \exists z. \ x \rightsquigarrow z \land z \rightsquigarrow y \)
- \( x \parallel y :\iff \exists z. \ x \rightsquigarrow z \land y \rightsquigarrow z \)

Moreover, we say that \( \rightsquigarrow \) is

- inductive iff for every sequence \( x_1 \rightsquigarrow x_2 \rightsquigarrow \ldots \rightsquigarrow x_i \rightsquigarrow \ldots \) there is an \( y \in \mathcal{E} \) with \( x_i \rightsquigarrow^* y \)
- acyclic iff there are no \( x, y \in \mathcal{E} \) with \( x \rightsquigarrow^+ y \) and \( y \rightsquigarrow^+ x \)
- noetherian iff there is no infinite sequence \( x_1 \rightsquigarrow x_2 \rightsquigarrow x_3 \rightsquigarrow \ldots \)
- strongly confluent iff for all \( x, y_1, y_2 \in \mathcal{E} \) with \( x \rightsquigarrow y_1 \) and \( x \rightsquigarrow y_2 \), there is a \( z \in \mathcal{E} \) with \( y_1 \rightsquigarrow^* z \) and \( y_2 \rightsquigarrow^* z \)
- weakly confluent\(^1\) iff for all \( x, y_1, y_2 \in \mathcal{E} \) with \( x \rightsquigarrow y_1 \) and \( x \rightsquigarrow y_2 \), there is a \( z \in \mathcal{E} \) with \( y_1 \rightsquigarrow^* z \) and \( y_2 \rightsquigarrow^* z \)
- confluent iff for all \( x, y_1, y_2 \in \mathcal{E} \) with \( x \rightsquigarrow^* y_1 \) and \( x \rightsquigarrow^* y_2 \) there is a \( z \in \mathcal{E} \) with \( y_1 \rightsquigarrow^* z \) and \( y_2 \rightsquigarrow^* z \)
- Church-Rosser iff for all \( x, y \in \mathcal{E} \) the property \( x \rightsquigarrow^* y \) is equivalent to \( x \parallel y \)

Note that \( \rightsquigarrow^* \) is reflexive and transitive, hence, a preorder. As every preorder, it can be interpreted as partial order on the equivalence classes of \( \rightsquigarrow \). However, if \( \rightsquigarrow \) is acyclic, then \( \rightsquigarrow^* \) is a partial order, since then \( x \approx y \) holds iff \( x = y \) holds.

Every noetherian relation is acyclic since otherwise a cycle \( x \rightsquigarrow^+ y \) and \( y \rightsquigarrow^+ y \) would generate infinite sequences \( x_1 \rightsquigarrow x_2 \rightsquigarrow x_3 \rightsquigarrow \ldots \). Moreover, every noetherian relation \( \rightsquigarrow \) is inductive, since there are only finite sequences \( x_1 \rightsquigarrow \ldots \rightsquigarrow x_n \) so that we have \( x_i \rightsquigarrow^+ x_n \). Noetherian relations are important since we can apply the principle of noetherian induction: If \( \rightsquigarrow \) is noetherian, and \( \forall x \in \mathcal{E} \). (\( \forall y \in \mathcal{E} \). \( x \rightsquigarrow^+ y \rightarrow \Phi(y) \)) \( \rightarrow \Phi(x) \) then we have \( \forall x \in \mathcal{E} \). \( \Phi(x) \).

The Church-Rosser property has been named after its inventors Alonzo Church and J. Barkley Rosser who proved that the \( \lambda \)-calculus has this property [69]. Note that \( x \rightsquigarrow^* y \) and \( y \rightsquigarrow^* x \) imply \( x \rightsquigarrow^* y \), but the converse is not necessarily the case. Note that \( x \rightsquigarrow^* y \) means that there are some elements \( z_1, \ldots, z_n \) such that \( x \rightsquigarrow z_1 \rightsquigarrow \ldots \rightsquigarrow z_n \rightsquigarrow y \). For this reason, it is trivial that \( x \parallel y \) implies \( x \rightsquigarrow^* y \), so that the interesting part of the Church-Rosser property is that \( x \rightsquigarrow^* y \) implies \( x \parallel y \). While \( \rightsquigarrow^* \) is directed, \( \rightsquigarrow^* \) is not directed and is therefore used to introduce a notion of equality.

\(^1\) Weak confluence is also often called local confluence.
Fig. 7.4. Equivalent Formulations of Different Forms of Confluence: In every case, for all $x, y_1, y_2$ the existence of $z$ is required.

The important property of confluence is that any sequence $x \rightsquigarrow^* y$ where $y$ cannot be further reduced, yields a unique normal form $y$ of $x$; If there would be another sequence $x \rightsquigarrow^* y'$ then by confluence, there must be a $z$ with $y \rightsquigarrow^* z$ and $y' \rightsquigarrow^* z$ which can only be satisfied by $y = z = y'$. Hence, even though there might be many ways to compute $y$ from $x$, we always obtain the same result $y$ in a confluent relation $\rightsquigarrow$. It is important to note that the Church-Rosser property and confluence as well as a slightly stronger form of confluence are all equivalent:

**Theorem 7.10 (Confluence).** The following are equivalent for any relation $\rightsquigarrow$:

(a) $\rightsquigarrow$ is confluent
(b) $\rightsquigarrow$ is Church-Rosser
(c) for all $x, y_1, y_2$ with $x \rightsquigarrow y_1$ and $x \rightsquigarrow^* y_2$, we have $y_1 \parallel y_2$

Proof. To prove the equivalence of (a) and (b), we first assume that $\rightsquigarrow$ is Church-Rosser, and assume $x \rightsquigarrow^* y_1$ and $x \rightsquigarrow^* y_2$. Then, we also have $y_1 \rightsquigarrow^* y_2$, so that by the Church-Rosser property of $\rightsquigarrow$, we conclude that $x \parallel y$ holds. Hence, $\rightsquigarrow$ is confluent. For the converse, we prove by induction on $n$ that for any confluent relation $\rightsquigarrow$, the property $x \rightsquigarrow^n y$ implies $x \parallel y$. The induction base is trivial, and in the induction step, we first assume (1) $x \rightsquigarrow^n y'$ and (2) $y' \rightsquigarrow y$. By the induction hypothesis, we know that there is a $z$ such that (3) $x \rightsquigarrow^* z$ and (4) $y' \rightsquigarrow^* z$ holds. We consider two cases derived from (2): If we have (2') $y \rightsquigarrow y'$, then we conclude from $y \rightsquigarrow y' \rightsquigarrow^* z$ and (3) that also $x \parallel y$.
Lemma 7.11 (Strong Confluence). The relation $\rightarrow$ is strongly confluent iff for all $x, y_1, y_2 \in E$ with $x \rightarrow^* y_1$ and $x \rightarrow^* y_2$, there is a $z \in E$ with $y_1 \rightarrow^* z$ and $y_2 \rightarrow^* z$.

Proof. Assume first that for all $x, y_1, y_2 \in E$ with $x \rightarrow^* y_1$ and $x \rightarrow^* y_2$, there is a $z \in E$ with $y_1 \rightarrow^* z$ and $y_2 \rightarrow^* z$. Since $x \rightarrow y_1$ and $x \rightarrow y_2$ imply $x \rightarrow^* y_1$ and $x \rightarrow^* y_2$, there must be a $z \in E$ with $y_1 \rightarrow^* z$ and $y_2 \rightarrow^* z$ (which implies $y_2 \rightarrow^* z$). Hence, $\rightarrow$ is strongly confluent.

Assume now that $\rightarrow$ is strongly confluent, i.e., for all $x, y_1, y_2 \in E$ with $x \rightarrow y_1$ and $x \rightarrow y_2$, there is a $z \in E$ with $y_1 \rightarrow^* z$ and $y_2 \rightarrow^* z$. We will now prove by induction on $n$ that for all $x, y_1, y_2 \in E$ with $x \rightarrow^* y_1$ and $x \rightarrow^* y_2$, there is a $z \in E$ with $y_1 \rightarrow^* z$ and $y_2 \rightarrow^* z$. For $n = 0$, we therefore have to prove that for all $x, y_1 \in E$ with $x \rightarrow^* y_1$, there is a $z \in E$ with $y_1 \rightarrow^* z$ and $x \rightarrow^* z$. This is easily seen by choosing $z := y_1$, since then $y_1 \rightarrow^* y_1$ holds by definition of $\rightarrow^*$ and $x \rightarrow^* y_1$ is one of our assumptions. For the induction step, assume we have $(1) x \rightarrow^* y_1$, $(2) x \rightarrow y_2$, and $(3) y_2' \rightarrow^* y_2$. $(1)$ means that either $x = y_1$ or $x \rightarrow y_1$ holds. If $x = y_1$ holds, choose $z := y_2$ to see that $y_1 = x \rightarrow^* z = y_2$ and $y_2 \rightarrow^* z = y_2$ hold. Otherwise, we have $(1') x \rightarrow y_1$. Since $\rightarrow$ is strongly confluent, we derive from $(1')$ and $(2)$ that there is a $z' \in E$ with $(4) y_1 \rightarrow^* z'$ and $(5) y_2' \rightarrow^* z'$. By induction hypothesis, we therefore derive from $(5)$ and $(3)$ that there is a $z \in E$ with $(6) z' \rightarrow^* z$ and $(7) y_2 \rightarrow^* z$.

Clearly, the notions of strong and weak confluence are not given by accident: We will prove next that strong confluence implies confluence, and confluence implies weak confluence, which is simple by the so-far obtained results:

Theorem 7.12 (Strong and Weak Confluence). Strong confluence implies confluence, and confluence implies weak confluence.

Proof. Assume $\rightarrow$ is strongly confluent, and assume that $x \rightarrow y_1$ and $x \rightarrow^* y_2$ hold. According to Theorem 7.10, we have to prove that there is a $z$ with...
Having considered various forms of confluence, we are now

Lemma 7.13 (Weak Confluence vs. Confluence). A noetherian relation $\leadsto$ is confluent iff it is weakly confluent. The converse is in general not true.

Proof. It is clear that confluence implies weak confluence. For the converse, we prove for every noetherian relation $\leadsto$ that for all $x, y, z$ with $x \leadsto^* y_1$ and $x \leadsto^* y_2$ that $y_1 \parallel y_2$. This is proved by noetherian induction: To this end, assume $x \leadsto^m y_1$ and $x \leadsto^m y_2$, and we will prove that there is a $z$ with $y_1 \leadsto^* z$ and $y_2 \leadsto^* z$. If $m = 0$, choose $z := y_2$, and if $n = 0$, choose $z := y_1$ to prove this. Otherwise (consider Figure 7.5), we have (1) $x \leadsto y_1$, (2) $y_1' \leadsto y_1$, (3) $x \leadsto y_2'$, and (4) $y_2' \leadsto y_2$. By weak confluence, we conclude from (1) and (3) that there exists $z'$ with (5) $y_1' \leadsto z'$ and (6) $y_2' \leadsto z'$. By our induction hypothesis, we conclude from (2) and (5) that there exists $z''$ with (7) $y_1 \leadsto z''$ and (8) $z' \leadsto z''$. By (6) $y_2' \leadsto z'$ and (8) $z' \leadsto z''$, we conclude (9) $y_2' \leadsto z''$. A further application of the induction hypothesis on (9) and (4) yields now that there exists $z$ with (10) $z'' \leadsto z$ and (11) $y_2 \leadsto z$. Combining (7) and (9) yields now (11) $y_1 \leadsto z'' \leadsto z$, so that by (10) and (11), we have $y_1 \parallel y_2$.

Weak confluence may not imply confluence in case that $\leadsto$ is not noetherian. An example given by Newman is as follows (see also [127]): assume elements $x, y_1, y_2$ and $y_1^i, y_2^i$ for $i \in \mathbb{N}$ with the following relation: $x \leadsto y_1$, $x \leadsto y_2$, $y_1^i \leadsto y_1$, $y_1^i \leadsto y_2^{i+1}$ as well as $y_1^i \leadsto y_1^i$ and $y_2^i \leadsto y_2^i$ for any $i \in \mathbb{N}$. It is then easily seen that $\leadsto$ is weakly confluent, but not confluent, since $x \leadsto y_1$ and $x \leadsto y_2$ holds, but there is no $z$ with $y_1 \leadsto z$ and $y_2 \leadsto z$.

Newman’s original proof [183] does not make use of noetherian induction, and is therefore more complicated.

7.1.4 Operational Semantics of DPNs

Todo! Having considered various forms of confluence, we are now able to discuss the right requirements for an operational semantics. Weak confluence is very important since it can be easily checked in contrast to confluence: In general, it may be possible that two different firing rules can be
applied to the same input stream and that these two firing rules may generate
different outputs. In such a situation, we have to check whether the two states
that are reached by the two transitions will meet later in a state, i.e., whether
the DPN is weakly confluent.

The first step in this analysis consists in determining critical pairs, i.e.,
states \( E_1 \) and \( E_2 \) that can be reached with one transition from the same state
\( E_0 \), i.e., \( E_0 \xrightarrow{N} E_1 \) and \( E_0 \xrightarrow{N} E_2 \). To this end, we have to compare the patterns
specified by different firing rules and have to find out whether they overlap,
i.e., whether we can replace their variables by expressions so that both pat-
tterns become the same. For term rewrite systems and automated theorem
proving, the concept of unification \([175, 200, 218]\) has been developed that
is useful for our purpose as well:

**Definition 7.14 (Most General Unifier).** Given two lists of patterns \([P_1, \ldots, P_n]\)
and \([P'_1, \ldots, P'_n]\), or two patterns \( P_0 \), \( P'_0 \), or two expressions \( e_i \), \( e'_i \), we compute
their most general unifiers \( \text{UnifyPs}(\[P_1, \ldots, P_n\], \[P'_1, \ldots, P'_n\]) \), \( \text{UnifyP}(P_0, P'_0) \),
and \( \text{UnifyE}(e_i, e'_i) \) recursively by the following rules:

- \( \text{UnifyPs}(\[\], \[\]) = \{\} \)
- \( \text{UnifyPs}(\[P_1, \ldots, P_n\], \[P'_1, \ldots, P'_n\]) = \varrho_1 \circ \varrho_2 \) where \( \varrho_1 := \text{UnifyP}(P_0, P'_0) \)
  and \( \varrho_2 := \text{UnifyPs}(\varrho_1(P_2), \ldots, \varrho_1(P_n), \varrho_1(P'_2), \ldots, \varrho_1(P'_n)) \)
- \( \text{UnifyP}(L, \sigma) = \{(L, \sigma)\} \)
- \( \text{UnifyP}(\sigma, L) = \{(L, \sigma)\} \)
- \( \text{UnifyP}(\[\], \[\]) = \varrho \)
- \( \text{UnifyP}((e :: \sigma), (e' :: \sigma')) = \varrho_1 \circ \varrho_2 \) where \( \varrho_1 := \text{UnifyE}(e, e') \)
  and \( \varrho_2 := \text{UnifyP}(\sigma, \sigma') \)
- \( \text{UnifyE}(x, e) = \{(x, \varrho(e))\} \) if \( x \in V \)
- \( \text{UnifyE}(e, x) = \{(x, \varrho(e))\} \) if \( x \in V \)
One can easily prove that the definition above computes the most general unifier that exists whenever a unifier exists at all (since our patterns are only first order expressions [175, 200]). Using unification, i.e., the computation of the most general unifier of two given patterns, we can find out whether they overlap, i.e., whether there is an input stream where these rules can be both applied. Using unification, we even obtain the most general of these input streams, and we can check whether the outputs are also different in this case. If they are different, we have a critical pair which cannot be rewritten anymore to become equal again, since outputs that were once generated will never be changed again. We therefore have the following simple theorem:

**Definition 7.15 (Overlapping and Critical Firing Rules).** Given firing rules for $m$ inputs and $n$ outputs over $\text{Stream}^m_D$ with variables $\mathcal{V}$:

\[
\begin{array}{c|c|c}
    x_1 & \cdots & x_m \\
    \hline
    y_1 & \cdots & y_n \\
    \hline
    P_{1,1} & \cdots & P_{1,m} & \sigma_{1,1} & \cdots & \sigma_{1,n} \\
    \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
    P_{p,1} & \cdots & P_{p,m} & \sigma_{p,1} & \cdots & \sigma_{p,n} \\
\end{array}
\]

Two firing rules with patterns $[P_{i,1}, \ldots, P_{i,m}]$ and $[P_{j,1}, \ldots, P_{j,m}]$ overlap with substitution $\varrho$ if $[\varrho(P_{i,1}), \ldots, \varrho(P_{i,m})] = [\varrho(P_{j,1}), \ldots, \varrho(P_{j,m})]$ holds. Two firing rules that overlap with $\varrho$ are called critical if we have $\varrho(\sigma_{i,k}) \neq \varrho(\sigma_{j,k})$ for at least one $k \in \{1, \ldots, n\}$.

Clearly, if there are two critical firing rules, then there is an input stream where the corresponding two rules can be applied and where these two rules produce different outputs. Hence, the existence of critical firing rules implies that the operational semantics does not define a function. Conversely, it is trivial that a function is obtained if there were no critical pairs. Hence, we have the following simple theorem:

**Theorem 7.16 (Critical Firing Rules).** A DPN is weakly confluent iff for every critical pair $(\mathcal{E}_1, \mathcal{E}_2)$, we have $\mathcal{E}_1 \parallel \mathcal{E}_2$

While the above holds for rewrite systems, it is not the case for DPNs due to the incremental computation.

**Theorem 7.17 (Critical Firing Rules).** A DPN according to Definition 7.2 whose base functions are defined via firing rules describes a stream processing function $\text{Stream}^m_D \rightarrow \text{Stream}^n_D$ iff there are no critical firing rules.

With the additional requirement that there are no critical firing rules, one might think that the operational semantics of a DPN can be easily described

---

2 Most general is meant here in the following sense: The set of expressions is partially ordered in that $e_1 \preceq e_2$ holds iff there is a substitution $\varrho$ such that $\varrho(e_1) = e_2$. Hence, variables are the least elements and constants the greatest elements in that partial order.
by simply firing the nodes as often as necessary (we allow infinite firings) to produce the output stream while consuming the input stream. However, this is not that easy as there is an additional problem that stems from the \textit{incremental computation of the output streams in the operational semantics} which arises if the outputs of one DPN node are the input of another node. In that case, the second node may already fire if sees a prefix of its input stream that matches with one of its patterns, but it may later on recognize that this was wrong.

To discuss this subtle problem in detail, we consider the following three functions \textit{merge}, \textit{por}, and \textit{cpy2}:

\begin{align*}
\text{merge} & \begin{array}{c|cc|c}
  x_1 & x_2 & y \\
  (a :: A) & (b :: B) & [a, b] \\
  (b :: B) & [b] & [a] \\
  (a :: A) & [a] & [a, b] \\
\end{array} \\
\text{por} & \begin{array}{c|cc|c}
  x_1 & x_2 & y \\
  (a :: A) & (b :: B) & [a \lor b] \\
  (1 :: A) & [1] & [1] \\
  (1 :: B) & [1] & [1] \\
\end{array} \\
\text{cpy2} & \begin{array}{c|cc|c}
  x_1 & x_2 & y_1 y_2 \\
  (a :: A) & (b :: B) & [a] [b] \\
  [b] & [b] & [0] \\
  (a :: A) & [a] & [a] \\
\end{array}
\end{align*}

Consider first the functions that were specified by the above firing rules. It is easily seen that there are no overlapping firing rules, so that all three tables specify stream-processing functions if their input streams are given at once (and not incrementally). Given finite input streams \(a_0, \ldots, a_n\) and \(b_0, \ldots, b_n\), \(b_{n+1}, \ldots, b_{n+p}\) of possible different lengths, \textit{merge} computes \(a_0, b_0, a_1, b_1, \ldots, a_n, b_n, b_{n+1}, \ldots, b_{n+p}\), and similarly, \textit{merge} computes \(a_0, b_0, a_1, b_1, \ldots, a_n, b_n, a_{n+1}, \ldots, a_{n+p}\) for inputs \(a_0, \ldots, a_n, a_{n+1}, \ldots, a_{n+p}\) and \(b_0, \ldots, b_n\). Similarly, for given boolean input streams \(a_0, \ldots, a_n\) and \(b_0, \ldots, b_n, b_{n+1}, \ldots, b_{n+p}\) of possible different lengths, \textit{por} computes \(a_0 \lor b_0, a_1 \lor b_1, \ldots, a_n \lor b_n, 1, \ldots, 1\), where the suffix 1, . . . , 1 of the output after \(a_n \lor b_n\) is the greatest prefix of \(b_{n+1}, \ldots, b_{n+p}\) consisting of only the value 1. The analog computation is performed in case the first input stream is longer than the second one. Finally, \textit{cpy2} is simply the identity function on \((D^n)^2\).

At a first glance, it seems that all three function tables specify simply stream-processing functions \(f_{\text{merge}}\), \(f_{\text{por}}\), and \(f_{\text{cpy2}}\) without problems. However, \textit{merge} may work wrong if its input streams are incrementally given: To see this, consider the following DPN:

\[
\begin{align*}
\text{even} & \begin{array}{c|c|c}
  z_1 & z_2 & y_1 \\
  [i] & [i+2] & [i+2] \\
\end{array} \\
\text{odd} & \begin{array}{c|c|c}
  z_1 & z_2 & y_1 \\
  [i] & [i+2] & [i+2] \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\begin{cases}
  (y_1, z_1) = \text{even}(z_1) \\
  (y_2, z_2) = \text{odd}(z_2) \\
  y_3 = \text{merge}(y_1, y_2)
\end{cases}
\end{align*}
\]

The expectation is that \(y_3 = 0, 1, 2, 3, 4, 5, \ldots\) should be computed, and indeed, that can be really computed, e.g. using the following firing sequence:

\[
\begin{align*}
(y_1 &\mapsto [i]) \\
(y_2 &\mapsto [i]) \\
(y_1 &\mapsto [i]) \quad \text{odd, even, merge} \\
(z_1 &\mapsto [i]) \\
(z_2 &\mapsto [i]) \\
(y_1 &\mapsto [0]) \quad \text{odd, even, merge} \\
(y_2 &\mapsto [1]) \quad \text{odd, even, merge} \\
(y_1 &\mapsto [2]) \\
(y_2 &\mapsto [3]) \\
(y_1 &\mapsto [4]) \quad \text{odd, even, merge} \\
(y_2 &\mapsto [5]) \\
(y_1 &\mapsto [0, 1, 2, 3]) \\
(y_2 &\mapsto [0, 1, 2, 3]) \quad \text{odd, even, merge}
\end{align*}
\]
Clearly, other firing sequences are possible, which is not unusual for a DPN. However, we can see with the firing sequence below, that not all of the possible computations yield the same stream for $y_5$:

\[
\begin{pmatrix}
| y_1 \mapsto [ ] & | y_1 \mapsto [ ] & | y_1 \mapsto [ ] & | y_1 \mapsto [ ] & | y_1 \mapsto [ ] \\
| y_2 \mapsto [ ] & | y_2 \mapsto [ ] & | y_2 \mapsto [ ] & | y_2 \mapsto [ ] & | y_2 \mapsto [ ] \\
| y_3 \mapsto [ ] & | y_3 \mapsto [ ] & | y_3 \mapsto [ ] & | y_3 \mapsto [ ] & | y_3 \mapsto [ ] \\
| z_1 \mapsto [ ] & | z_1 \mapsto [ ] & | z_1 \mapsto [ ] & | z_1 \mapsto [ ] & | z_1 \mapsto [ ] \\
| z_2 \mapsto [ ] & | z_2 \mapsto [ ] & | z_2 \mapsto [ ] & | z_2 \mapsto [ ] & | z_2 \mapsto [ ] \\
\end{pmatrix}
\]

In general, a firing rule with a finite pattern, i.e., with $[ ]$ or $[e_1, \ldots, e_n]$ could erroneously match since the stream is not yet complete. We say ‘erroneously match’ since the matching would no longer be possible if the producer of this stream would add further values to it. Hence, if the consumer node will fire before the producer of its input stream adds more values to the stream, the computation can go wrong: Assume a node implements a stream processing function $f : \text{Stream}^m \rightarrow \text{Stream}^n$, and is applied to an input stream $\alpha_1\alpha_2\beta$, but currently only sees the prefix $\alpha_1\alpha_2$. Assume that there is a firing rule that can consume $\alpha_1$ from the prefix $\alpha_1\alpha_2$ by generating the output stream $f(\alpha_1)$. Whether this computation was impatiently performed by an erroneous match or not depends on the values $f(\alpha_1)$, $f(\beta)$, and $f(\alpha_1\alpha_2\beta)$: If the same firing rule could be applied not only to $\alpha_1\alpha_2$, but also to $\alpha_1\alpha_2\beta$ consuming also $\alpha_1$ and producing the same value $f(\alpha_1)$, the computation would be correct. Since the remainder of the input stream $\alpha_1\alpha_2\beta$ is then $\alpha_2\beta$, it moreover follows that $f(\alpha_1\alpha_2\beta) = f(\alpha_1) \cdot f(\alpha_2\beta)$ must hold that the computation is correct. However, if the firing rule can no longer be applied to $\alpha_1\alpha_2\beta$, the computation done by consuming only $\alpha_1$ from a prefix $\alpha_1\alpha_2$ without knowing the further suffix $\beta$ was obviously different. We therefore have already proved one half of the following theorem that gives us a criterion to allow arbitrary firings of the nodes in a dataflow network:

**Theorem 7.18 (Scheduling-Independence).** The schedule of firings of a node of a DPN implementing function $f$ is independent of the firings of its producer nodes iff for each input stream $\alpha\beta$ where $\alpha$ can be consumed by a firing rule of $f$ without the suffix $\beta$, we have $f(\alpha\beta) = f(\alpha) \cdot f(\beta)$.

**Proof.** Assume first that for each input stream $\alpha\beta$ where $\alpha$ can be consumed by a firing rule of $f$ without the suffix $\beta$, we have $f(\alpha\beta) = f(\alpha) \cdot f(\beta)$. Then, if the producer of this stream has already produced $\alpha$, the node may fire so that $\alpha$ is consumed and $f(\alpha)$ is produced. The remainder of the input stream is $\beta$, so that the following computations will consume $\beta$ by producing $f(\beta)$. Thus, the entire stream $\alpha\beta$ was consumed by producing $f(\alpha) \cdot f(\beta) = f(\alpha\beta)$ which is the correct function value.

Now assume that there is an input stream $\alpha\beta$ where $\alpha$ can be consumed by a firing rule of $f$ without the suffix $\beta$, but we have $f(\alpha\beta) \neq f(\alpha) \cdot f(\beta)$. Thus, a first schedule could generate $\alpha$ which is then consumed by firing $f$ producing $f(\alpha)$, so that the further computation consumes $\beta$ and produces $f(\beta)$. Thus,
we obtain by this schedule the output \( f(\alpha) \cdot f(\beta) \), while the schedule that will first generate the entire\(^3\) input stream \( \alpha\beta \) will produce \( f(\alpha\beta) \neq f(\alpha) \cdot f(\beta) \).

We can moreover easily prove the following relationship to monotonicity:

**Theorem 7.19 (Scheduling-Independence implies Monotonicity).**

If for each input stream \( \alpha\beta \) where \( \alpha \) can be consumed by a firing rule of \( f \) without the suffix \( \beta \), we have \( f(\alpha\beta) = f(\alpha) \cdot f(\beta) \), then \( f \) is monotone. The converse is in general not valid.

**Proof.** We have \( f(\alpha) \leq f(\alpha) \cdot f(\beta) = f(\alpha\beta) \), which means by Lemma A.39 that \( f \) is monotone. To see that the converse is in general not true, note that the function \( \text{por} \) as discussed before is monotone, but since \( \text{por}(([1],[1])) = [1] \neq [1,1] = \text{por}(([1],[1]) \cdot ([1],[1])) \) it is not scheduling independent. \( \square \)

Note that if a function is monotone, we know that \( f(\alpha) \leq f(\alpha\beta) \) holds by Lemma A.39. Thus, there is a \( \gamma \) so that \( f(\alpha\beta) = f(\alpha) \cdot \gamma \). However, it may not be the case that \( \gamma = f(\beta) \) holds (which is the case for \( \text{por} \)).

**Theorem 7.20 (Scheduling-Independence is Weak Confluence).**

A DPN \( N \) is scheduling-independent iff its firing relation \( \rightarrow_N \) is weakly confluent.

**Proof.** Assume we have \( \sigma = \alpha\beta = \alpha'\beta' \) and we have \( f(\alpha) \cdot f(\beta) = f(\sigma) = f(\alpha') \cdot f(\beta') \).

Since weak confluence is the same as confluence by Newman’s Lemma, we conclude that scheduling independent DPNs that are scheduling-independent are confluent so that their resulting output data stream is uniquely determined.

As already mentioned in the proof of the above theorem, the function \( \text{por} \) is a good example to demonstrate that even monotone functions may not be scheduling independent. To this end, consider the following DPN:

\[
\begin{array}{ccc}
\text{cpy1} & \text{por} & \\
\begin{array}{c}
\begin{array}{c}
 x_1 \quad y_1 \\
 \text{(a :: A)} \quad [a]
\end{array}
\end{array} & \\
\begin{array}{c}
\begin{array}{c}
 x_2 \\
 \text{(a :: A) [b :: B) [a v b)}
\end{array}
\end{array} & \\
\begin{array}{c}
\begin{array}{c}
 \begin{array}{c}
 1 :: A \quad [1]
\end{array}
\end{array}
\end{array} & \\
\begin{array}{c}
\begin{array}{c}
 \begin{array}{c}
 1 :: B \quad [1]
\end{array}
\end{array}
\end{array} & \\
\end{array}
\end{array}
\]

The expectation is that for the input stream \((x_1,x_2) = ([1,1],[1,1])\), the output \( y = [1,1] \) should be computed. For example, this is done by the following firing sequence:

\[^3\]There is no need to produce the entire stream \( \alpha\beta \), but a finite prefix of it that is larger than all the finite matches used in the firing rules.
scheduling-independent: Using $\alpha D$ and it is even continuous, since it is simply the identity function on scheduling-independent. $\text{cpy2}$ is scheduling-independent, thus also mono-

Looking carefully at the problem of scheduling-independence, we see that its

If $f$ does only have patterns of the $\alpha \beta$ without knowing about the suffix $\alpha$. If we forbid the finite patterns $[\ ]$ and $[e_1, \ldots, e_n]$, then we have the following result:

Theorem 7.21 (Prefix Matchings imply Scheduling-Independence).

If the firing rules of a function $f$ do not make use of the finite patterns $[\ ]$ and $[e_1, \ldots, e_n]$, then $f$ is scheduling-independent.

Proof. If $f$ can fire when it is applied to the input stream $\alpha$, then it can fire the same rule on the input stream $\alpha \beta$, since $f$ does only have patterns of the form $(e_1 :: \ldots :: (e_n :: L))$. For this reason, we have $f(\alpha \beta) = f(\alpha) \cdot f(\beta)$ for each input stream $\alpha \beta$ where $\alpha$ can be consumed by a firing rule of $f$ without the suffix $\beta$. Thus, $f$ is scheduling-independent. \[\square\]

We can show that $\text{por}$ is monotone, but as demonstrated above, it is not scheduling-independent. $\text{cpy2}$ is scheduling-independent, thus also monotone, and it is even continuous, since it is simply the identity function on $D^* \times D^*$. Finally, note that $\text{merge}$ is not even monotone, and therefore also not scheduling-independent: Using $\alpha := ([\ ], [b_0])$ and $\beta := ([a_0], [\ ])$, we have:

$$\text{merge}([\alpha \beta]) = [b_0] \not

Looking carefully at the problem of scheduling-independence, we see that its

If a function does not have patterns $[\ ]$ and $[e_1, \ldots, e_n]$, then it cannot test its input stream for finiteness, and instead can only test whether a suitable prefix is available for firing a rule. It is then pretty clear that the above theorem holds, since no matter what the suffix $\beta$ is the same firing rule can be applied, so that this rule is context-insensitive.
The above theorem is often stated as forbidding to test input buffers for emptiness.

To define the operational semantics, we consider variable environments $E : \mathcal{V}_\text{in} \cup \mathcal{V}_\text{out} \cup \mathcal{V}_\text{loc} \rightarrow \text{Stream}_n^\mathbb{R}$. Any base function is then described by means of transition rules $E \rightarrow E'$ that modify the values according to the firing rules and the equations of the DPN.

Finally, consider the following function [189] that is stable, but not sequential [189]:

$$
\begin{array}{cccc}
\text{x}_1 & \text{x}_2 & \text{x}_3 & y \\
\text{(1 :: A)} & \text{(0 :: B)} & \text{C} & 1 \\
\text{A} & \text{(1 :: B)} & \text{(0 :: C)} & 1 \\
\text{(0 :: A)} & \text{B} & \text{(1 :: C)} & 1 \\
\text{(0 :: A)} & \text{(0 :: B)} & \text{(0 :: C)} & 0 \\
\text{(1 :: A)} & \text{(1 :: B)} & \text{(1 :: C)} & 0 \\
\end{array}
$$

It is easily seen that is scheduling-independent, since it does not make use of finite patterns. It is, however, not sequential, since there is no input stream where in each of the firing rules a value is read, in other words, for each input stream, there is a firing rule that does not read from it. Hence, we see that there are scheduling-independent functions that are not sequential.

The converse is however true:

Another interesting function is the following slack-aware parallel-or:

$$
\begin{array}{cccccc}
\text{x}_1 & \text{z}_1 & \text{x}_2 & \text{z}_2 & y & \text{z}'_1 \text{z}'_2 \\
\text{(a :: A)} & \text{[ ]} & \text{(c :: C)} & \text{[ ]} & \text{[ ]} & \text{[ ]} \\
\text{(1 :: A)} & \text{[ ]} & \text{[ ]} & \text{D} & \text{[ ]} & \text{[ ]} \\
\text{[ ]} & \text{B} & \text{(1 :: C)} & \text{[ ]} & \text{[ ]} & \text{[ ]} \\
\text{(a :: A)} & \text{(b :: B)} & \text{C} & \text{D} & \text{[ ]} & \text{[ ]} \\
\text{A} & \text{B} & \text{(c :: C)} & \text{(d :: D)} & \text{[ ]} & \text{[ ]} \\
\end{array}
$$

The above function reads two boolean input streams $x_1$ and $x_2$, and generates as boolean output stream $y$ their disjunction. To this end, it also maintains two local streams $z_1$ and $z_2$ to remember a slack that can be obtained as follows: Initially, there is no slack, i.e., both local buffers $z_1$ and $z_2$ are empty. As long as boolean values can be read from both input streams, these are consumed, and their disjunction is written to the output stream $y$ according to the first firing rule. No slack is generated here, i.e., nothing is written to the local buffers. The second and third rules are responsible for generating a slack: If one of the input streams $x_i$ is empty, while the other one starts with 1, we generate the output 1 since that is true whatever value will be read from the other stream (also if nothing will be read from there since it has already been consumed). In addition to writing 1 to the output stream $y$, we also write this
value (but we could write any other value, too) to the local buffer $z_i$ that corresponds with the other input stream that is currently empty. If later on, a value should appear on that stream either rule four or five must be applied to consume that value and also one value of the corresponding local buffer until that local buffer becomes empty again.

Hence, the meaning of $z_i = \[]$ is that the next value read from $x_i$ should be considered for the next output, and having $n$ elements in $z_i$ means that the next $n$ elements of $x_i$ are irrelevant since the corresponding outputs have already been generated since the other streams had $n$ 1 values at these positions.

It can be seen that sapor is not sequential, since each column has a firing rule with the pattern $\[]$. It can however be easily proved that sapor is weakly confluent.

Much simpler are the following functions used in Boolean Dataflow networks since they are all sequential: The switch node reads a value from each of its input streams $x_1$ and $x_2$. If the value read from $x_1$ is true, it forwards the value read from the other stream to output $y_1$ otherwise to $y_2$. Depending on the boolean value read from $x_1$, the select node either reads a value from $x_2$ or $x_3$ and sends that value to its output stream. The meanings of upsample and downsample are clear.

\[
\begin{array}{ccc}
\text{switch} & y_1 & y_2 \\
x_1 & x_2 & \text{select} \\
(1 :: A) & (b :: B) & b \[ \\
(0 :: A) & (b :: B) & [b] \\
\text{upsample} & y_1 \\
(a :: A) & [a, a] \\
\text{downsample} & y_1 \\
(a :: (a :: A)) & [a] \\
\end{array}
\]

7.1.5 The Full Abstraction Problem

**Theorem 7.22 (Adequacy of Denotational Semantics).** Given a DPN $N$, and a pattern-consistent interpretation $I$ as well as its state transition diagram $T_N$.

For every $(u_1, \ldots, u_m) \in \text{Stream}_m^D$, there is by definition of $T_N$ an initial state $E_0$ with $E_0(x_i) := u_i$ (and $E_0(x) := \[]$ for $x \notin V_m$). For this initial state $E_0$, there is a final state $E_f$ so that $E_0 \rightsquigarrow E_f$ and the following holds:

$$
(E_f(y_1), \ldots, E_f(y_n)) = \llbracket N \rrbracket_I (E_0(x_1), \ldots, E_0(x_m))
$$

The above theorem states that the denotational semantics can be found in the operational semantics, i.e., there is a schedule to fire the nodes of a DPN such that the operational semantics computes the values defined by the denotational semantics.
By the examples with the merge and the par nodes, we know that the converse is not true in general, since the denotational semantics defines also functions for these DPNs. However, we have the following:

**Theorem 7.23 (Fully Abstract Denotational Semantics).** *Given a DPN $N$ whose state transition diagram $T_N$ has a confluent transition relation $\leadsto$. Then, for every initial state $E_0$ and its uniquely determined final state $E_f$, we have the following relationship for any pattern-consistent interpretation $I$:

$$(E_f(y_1), \ldots, E_f(y_n)) = \llbracket N \rrbracket_I (E_0(x_1), \ldots, E_0(x_m))$$

extend the above theorems to infinite streams!*

There is a strong relationship to the denotational semantics of functional languages [39, 178, 188, 189, 208].

While $\perp$ is used to denote in a functional program that the evaluation of an expression does not lead to a valid argument, it means for a dataflow network that the corresponding input buffer is empty. This is close relationship provided that we assume that the dataflow networks work in a output-driven manner, i.e., that the nodes ask via their inputs for computing corresponding input values. This is similar to the lazy evaluation of argument expressions of a functional program [103, 124].

<table>
<thead>
<tr>
<th>parallel – or</th>
<th>sequential – or</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp$ 0 1</td>
<td>$\perp$ 0 1</td>
</tr>
<tr>
<td>$\perp$ $\perp$ 1</td>
<td>$\perp$ $\perp$ 1</td>
</tr>
<tr>
<td>0 $\perp$ 0 1</td>
<td>0 $\perp$ 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

For example, here is an example of the common effects: Assume we wish to implement the boolean disjunction by means of an if-then-else operation. To compute $\varphi \lor \psi$ for expressions $\varphi$ and $\psi$, a functional program could try to first evaluate $\varphi$. Depending on the result of this evaluation, we have three cases:

- If the result returned is 1, the entire result of the evaluation of $\varphi \lor \psi$ is 1, and $\psi$ is not evaluated.
- If the result returned is 0, the final result is the result of the evaluation of $\psi$.
- If the result returned is $\perp$, i.e., the evaluation failed or does not terminate, then this is also the result of the evaluation of $\varphi \lor \psi$.

The problem is thereby that due to the sequential nature of the functional language considered, we either first evaluate $\varphi$ or $\psi$. For this reason, the thereby implemented disjunction is not commutative, in contrast to a parallel-or. In contrast, there is no problem to use the parallel-or as a basic function, since the parallel-or is a continuous function (it is monotone, and $\{\perp, 1, 0\}$ is finite).
For this reason, the operational semantics will differ from the operational semantics which has been observed by Milner and Plotkin [178, 208]. They therefore posed the full abstraction problem which consists in determining a subset of the continuous functions that can also be implemented via the operational semantics. To this end, sequential functions have been defined [56, 178, 258]:

**Definition 7.24 (Sequential Function).** A continuous function \( f \) with \( n \) arguments is sequential if it is either a constant function or there is an index \( i \) such that the following holds:

- \( x_i = \perp \) implies \( f(x_1, \ldots, x_m) = \perp \) for all possible argument values \( x_i \)
- for every \( d \in D \), the functions \( f(x_1, \ldots, x_{i-1}, d_i, x_{i+1}, \ldots, x_m) \) are sequential

In this case, we say \( i \) is a primal index of \( f \).

Hence, by definition there is a sequential algorithm to evaluate a sequential function. We first evaluate the argument whose index is the primal index, and then go on by evaluating the obtained co-factored functions \( f(x_1, \ldots, x_{i-1}, d_i, x_{i+1}, \ldots, x_m) \) according to their primal indices.

The parallel-or function is therefore not sequential, but continuous, and therefore there are continuous functions that are not sequential. Plotin shows that the Scott-Milner model is fully abstract if the parallel-or function is used as a basic function [208].

However, even using sequential functions, the full abstraction problem has not been solved. For this reason, Berry defined a further subset called stable functions [39, 44, 189].

In the previous section, we have defined a denotational semantics for DPN that assumes that we can already assign to each basic function \( f_i \) a continuous function \( \mathcal{I}(f_i) \).

Consider examples:

- parallel-or, sequential-or
- select, switch, if-then-else
- merge
- Lustre/SIGNAL: when, current, default
- dataflow network with continuous functions that requires unbounded memory:

\[
\text{local}(\{z_1, z_2, z_3\}, \begin{cases} (z_1, z_2) = \text{cpy}(x) \parallel \\ z_3 = \text{even}(z_2) \parallel \\ y = z_2 + z_3 \end{cases})
\]

The meaning of \( \text{cpy} \) is that the input value \( x \) is copied both to \( z_1 \) and \( z_2 \), and the meaning of \( \text{even} \) is that two values are read from buffer \( z_2 \) and the second one is copied to \( z_3 \). The meaning of \( y \) is clear.
The Kahn principle essentially means that the operational and denotational semantics coincide.

7.2 Important Classes of Dataflow Process Networks

In this section, we consider instances of dataflow process networks that became successful in practice. It is remarkable that all of them do not take advantage of the possibility to control the execution of the DPN in a data or demand-driven fashion [103, 124, 205, 206, 254]. Instead, the clock-driven execution is preferred that can come in two forms: First, static periodic schedules can be defined for static DPNs that are discussed in the following three sections that cover synchronous dataflow and cyclo-static dataflow. The static periodic schedules that are computed at compile time for these DPNs, can be viewed as multiple clocks since there is a slowest clock that ticks at the frequency given by the period of the overall static schedule, and within this period, each node has its own subclock that might trigger the node more than once within such a period. Hence, these systems are already multicontrol systems or polychronous systems. These approaches become very successful, in particular, for the synthesis of signal processing systems [5, 47, 106, 108, 160, 161, 182, 214].

Second, even though a static schedule might not exist due to dynamic data dependencies, clocks may nevertheless be defined. This is typically done in the languages Lustre [35, 66, 119, 120] and Signal [29, 35, 107, 155–157] that impose clock constraints on the data streams so that a polychronous execution can be run with bounded memory requirements.

Another way to make use of the algorithms for static DPNs is to consider quasi-static DPNs [106, 117] where dynamic DPNs are partitioned so that the DPN obtained by collapsing each partition into a single process node is a static DPN. Then, a static periodic schedule can be computed for this DPN and the scheduling problem is reduced to the smaller ones in the partitions. While this might appear as an ad hoc approach is work well in practice, since the DPNs generated by statements of structured programming languages without goto statements allows one to apply this approach.

7.2.1 Single-Rate Dataflow Process Networks

The simplest kind of DPNs are obtained if we restrict the firing rules for every base function as follows:

\[
\begin{array}{c|c}
  x_1 & \ldots & x_m \\
  (a_1 :: A_1) \ldots (a_m :: A_m) \\
  \hline
  y_1 & \ldots & y_n \\
  f_1(a_1, \ldots, a_m) \ldots f_1(a_n, \ldots, a_m) \\
\end{array}
\]
This means that each firing of a node consumes from each of its input streams exactly one value and produces on each of its output streams exactly one value. For this reason, it is obvious that all nodes can synchronously fire. For this reason, one can fire single-rate DPNs with a single clock, and as soon as all buffers were filled with a single data value, we can maintain this property as an invariant of the execution of the entire DPN.

Synchronous languages without causality problems can be easily translated into equivalent single-rate DPNs. To this end, one can generate the equation based code so that each equation is mapped to one node of the DPN that will produce the value of a local or output variable. Since these DPNs are a strict subset of the synchronous dataflow process networks that are discussed in the following section, we do not consider them in more details, and directly proceed to the more general class.

### 7.2.2 Synchronous Dataflow Process Networks

A less restricted class of DPNs is obtained by the synchronous dataflow networks (SDF) of Lee and Messerschmitt [46, 47, 60, 159, 162–164]. The firing rules are thereby restricted as follows: each base function has only a single firing rule whose patterns are of the form

\[
(a_{1,1} :: \cdots (a_{1,c_1} :: A_1)) \cdots (a_{m,1} :: \cdots (a_{m,c_m} :: A_m) )
\]

where the sequences \( \sigma_i \) are allowed as usual to arbitrarily use the matched values \( a_{i,j} \). Hence, for each input stream \( x_i \) and each output stream \( y_i \), there are constants \( c_i \) and \( p_i := |\sigma_i| \), respectively, so that in each firing step, the node always consumes \( c_i \) data values of input stream \( x_i \) and produces \( p_i \) values for output stream \( y_i \).

Hence, the production and consumption of data values of process nodes in SDF is static, i.e., independent of particular data values. For this reason, SDFs are also often called static DPN. Having only static process nodes, it is not surprising that static analysis is possible to solve relevant questions for these DPNs: We will now show that for SDF, one can statically determine whether bounded FIFO buffers were sufficient to run the DPN infinitely long.

To this end, we write \( \text{prod}(z_i) \) and \( \text{cons}(z_i) \) for the producer and consumer nodes of a local variable \( z_i \in V_{\text{loc}} \), and \( p(z_i) \) and \( c(z_i) \) for the number of data values produced and consumed by \( \text{prod}(z_i) \) and \( \text{cons}(z_i) \), respectively. If the number of data values contained in \( z_i \) should be bounded, the number of tokens produced and consumed must be equal on the long run. Otherwise, the buffer would become empty or will grow over any bound. Hence, if there should be a static periodic schedule where node \( f_i \) fires \( NF(f_i) \) many times in each period, then it satisfies the following balance equations for every local variable \( z_i \):
Theorem 7.25 (Balance Equations). Let \( n \) be a synchronous DPN, where \( \text{prod}(z_i) \) and \( \text{cons}(z_i) \) denote the producer and consumer of the local variable \( z_i \in \mathcal{V}_{\text{loc}} \) and \( p(z_i) \) and \( c(z_i) \) are the number of values produced and consumed by \( \text{prod}(z_i) \) and \( \text{cons}(z_i) \), respectively. If there is a static periodic schedule where \( \text{NF}(f) \in \mathbb{N} \) is the number of times node \( f \) is fired in each period, then the following balance equation must hold for each local variable \( z_i \):

\[
\text{prod}(z_i) \cdot \text{NF}(z_i) = \text{cons}(z_i) \cdot \text{NF}(z_i)
\]

If the resulting linear equation system should only have the trivial solution \( \text{NF}(f) = 0 \) for each node \( f \), then there is no such static periodic schedule. Otherwise, each nontrivial solution yields the numbers of firings \( \text{NF}(f) > 0 \) node \( f \) has to perform in one period of the static schedule.

For example, consider the DPN \( P_1 \) shown on top in Figure 7.6. We obtain the following balance equations for the four local variables \( z_1, z_2, z_3, z_4 \):

\[
\begin{align*}
(z_1) & : 1 \cdot \text{NF}(f_1) = 2 \cdot \text{NF}(f_2) \\
(z_2) & : 3 \cdot \text{NF}(f_3) = 1 \cdot \text{NF}(f_1) \\
(z_3) & : 3 \cdot \text{NF}(f_3) = 2 \cdot \text{NF}(f_2) \\
(z_4) & : 2 \cdot \text{NF}(f_2) = 3 \cdot \text{NF}(f_3)
\end{align*}
\]

Using the balance equations, we obtain a linear equation system that can be solved for the number of firings of each node in the static schedule. In the above example, all solutions are \((\text{NF}(f_1), \text{NF}(f_2), \text{NF}(f_3)) = (6, 3, 2)\lambda\) for every \( \lambda \in \mathbb{R} \). Hence, if a static schedule should exist that can be infinitely often repeated without requiring infinite local memory, then the number of times nodes \( f_1, f_2, \) and \( f_3 \) fire must be \( 6\lambda, 3\lambda, \) and \( 2\lambda \), respectively for \( \lambda \in \mathbb{N} \).

Hence, we have found a simple criterion to check whether a static schedule may exist. However, note that even if the balance equations have a nontrivial solution, it is not guaranteed that also a static periodic schedule exists. The only reason for this can be that there are not enough data values available to execute the required number of firings of the process nodes. The above result can therefore be slightly sharpened in that we state that if the balance equations have non-trivial solution, then there always exists a static periodic schedule with an initially available minimal amount of data values.

For example, the balance equations of the second DPN \( P_2 \) shown in Figure 7.6 has the nontrivial solutions \( \text{NF}(f_1) = 3p, \text{NF}(f_2) = 2p, \) and \( \text{NF}(f_3) = p, \) so that we known that if a static periodic schedule exists, then node \( f_1 \) must fire 3 times as often as node \( f_3 \) and node \( f_2 \) must fire two times as often as \( f_3 \). However, whether a static periodic schedule exists or not depends also on the initial values of the variables. A possible schedule could look as follows (the numbers denote the number of values contained in the local variables \( z_1, z_2, \) and \( z_3 \)).
Fig. 7.6. Four Static DPNs Modeled by their Graphical Representations, their Equation Systems, and their Incidence Matrices (see Definition 7.26)
It is clear that no schedule exists if we would like to start with \((0, 0, 1)\) instead, since none of the nodes would be enabled in that state. In particular, we can start in any state that is obtained by taking one state of the above schedule and increasing the numbers contained in it.

In contrast, the balance equations of DPN \(P_4\) shown in Figure 7.6 has only the trivial solution \(\text{NF}(C) = \text{NF}(D) = \text{NF}(P) = 0\). Therefore, we conclude that there is no static periodic schedule that could run \(P_4\) with bounded memory requirements. The problem is that \(D\) requires two data values produced by \(C\) in order to produce one data value in \(z_3\), but then \(C\) produces also two data values in \(z_2\). Since \(P\) only consumes one value from \(z_3\) and \(z_4\) when it fires, the number of data values to be stored in \(z_2\) can not be bounded.

As the single nodes in a synchronous DPN fire with different rates, they are also sometimes called \textit{multirate DPNs}, to show up the difference to the single-rate DPNs considered in the previous subsection. Hence, the solutions of the balance equations give us also information about all the possible static periodic schedules.

Note further that \textit{strongly connected DPNs} will always lead to an incidence matrix \(I_P\) whose null space has at most dimension 1, since the buffer that connects different strongly connected components imposes constraints on the number of firings of the nodes in these components. Consider the DPN \(P_3\) whose buffer \(z_3\) imposes such a constraint.

A weakness of the static scheduling of synchronous DPNs is that the period of the static schedule can become very large. In the worst case, it may have a size that is exponential in the number of nodes of the DPN. As such a worst case example, consider the following DPN:

\[
\begin{array}{c}
\circ \quad f_0 \quad m \quad \circ \quad f_1 \quad 1 \quad \circ \quad f_2 \quad m \quad \circ \quad f_3 \quad 1 \quad \circ \quad f_4 \quad 1 \quad \cdots
\end{array}
\]

Using \(n\) such nodes \(f_0, \ldots, f_{n-1}\) yields the balance equations \(m \cdot \text{NF}(f_0) = \text{NF}(f_1), m \cdot \text{NF}(f_1) = \text{NF}(f_2), \ldots, m \cdot \text{NF}(f_{n-2}) = \text{NF}(f_{n-1})\), so that \(\text{NF}(f_i) = m_i \cdot \text{NF}(f_0)\) holds.

For example, the balance equations of DPN \(P_3\) of Figure 7.6 has the solutions \(\text{NF}(f_1) = 4p, \text{NF}(f_2) = 8p, \text{NF}(f_3) = 2p, \text{NF}(f_4) = p\), so that 8 invocations of \(f_2\) must be included in one period of the static periodic schedule.

As the number of firings of the nodes might become quite large in a periodic schedule, one has to carefully design the schedules themselves. It has already been observed in [46] that this is an important issue for the efficiency of the generated code. Further investigations of the problem to find good schedules for the period of a static periodic schedule can be found in [47, 48, 182].

Since the particular data values are irrelevant for the consideration of memory requirements of static DPNs, and instead, only the numbers of data values produced and consumed are of interest, the state transition system of static DPNs can be abstracted as follows:
Definition 7.26 (Markings and Incidence Matrix of a DPN). The incidence matrix of a DPN \( P \) with \( m \) buffers \( z_1, \ldots, z_m \) and \( n \) process nodes \( f_1, \ldots, f_n \) is a \( m \times n \) matrix \( \mathcal{I}_P \) whose entries \( t_{i,j} \) are the number of tokens that are produced (or consumed if negative) in buffer \( z_i \) when process node \( f_j \) fires, i.e.

\[
  t_{i,j} := \begin{cases} 
    p(z_i) - c(z_i) & : \text{if } f_j = \text{prod}(z_i) = \text{cons}(z_i) \\
    p(z_i) & : \text{if } f_j = \text{prod}(z_i) \neq \text{cons}(z_i) \\
    -c(z_i) & : \text{if } f_j = \text{cons}(z_i) \neq \text{prod}(z_i) \\
    0 & : \text{otherwise}
  \end{cases}
\]

A marking \( v = (v_1, \ldots, v_m) \) of \( P \) is a \( m \)-vector of natural numbers \( v_i \in \mathbb{N} \) that encodes the number of data values contained in buffer \( z_i \) in a particular state of \( P \).

Figure 7.6 already contained the incidence matrices of the four listed DPNs. Note that the incidence matrix does not uniquely represent the DPN, since a process node \( f_j \) that produces and consumes the same number of data values from the same buffer \( z_i \) will yield entry \( t_{i,j} = p(z_i) - c(z_i) \) such that the numbers \( p(z_i) \) and \( c(z_i) \) cannot be read from the entry \( t_{i,j} \).

The balance equations can thus be expressed by means of the incidence matrix \( \mathcal{I}_P \): It is easily seen that the linear equation system imposed by the balance equation is \( \mathcal{I}_P \cdot r = 0 \). Using the incidence matrix \( \mathcal{I}_P \) of a static DPN \( P \), one can moreover check the non-reachability of markings, since we have the following results [89, 90, 181]:

Theorem 7.27 (Reachability and Invariants). Given a DPN \( P \) with \( m \) buffers \( z_1, \ldots, z_m \) and \( n \) process nodes \( f_1, \ldots, f_n \), the following holds:

- If a marking \( v = (v_1, \ldots, v_m) \) can be reached from a marking \( u = (u_1, \ldots, u_m) \), then there is a firing vector \( r = (r_1, \ldots, r_n) \) where \( v_i, u_i, r_i \in \mathbb{N} \) such that for the incidence matrix \( \mathcal{I}_P \), we have \( \mathcal{I}_P \cdot r = v - u \), i.e.

\[
\begin{pmatrix}
  t_{1,1} & \ldots & t_{1,n} \\
  \vdots & \ddots & \vdots \\
  t_{m,1} & \ldots & t_{m,n}
\end{pmatrix}
\begin{pmatrix}
  r_1 \\
  \vdots \\
  r_n
\end{pmatrix}
= 
\begin{pmatrix}
  v_1 - u_1 \\
  \vdots \\
  v_m - u_m
\end{pmatrix}
\]

The firing vector \( r = (r_1, \ldots, r_n) \) contains the number of firings \( r_i \) of node \( f_i \) for the firing sequence from \( u \) to \( v \).

- Any solution \( w = (w_1, \ldots, w_m) \) of the linear equation system \( \mathcal{I}_P^\top \cdot w = 0 \) where \( \mathcal{I}_P^\top \) is the transpose of \( \mathcal{I}_P \), i.e.,

\[
\begin{pmatrix}
  t_{1,1} & \ldots & t_{m,1} \\
  \vdots & \ddots & \vdots \\
  t_{1,n} & \ldots & t_{m,n}
\end{pmatrix}
\begin{pmatrix}
  w_1 \\
  \vdots \\
  w_m
\end{pmatrix}
= 
\begin{pmatrix}
  0 \\
  \vdots \\
  0
\end{pmatrix},
\]

is called an invariant of the DPN \( P \). If a marking \( v = (v_1, \ldots, v_m) \) can be reached from a marking \( u = (u_1, \ldots, u_m) \), then for every invariant \( w = \)
(w_1, \ldots, w_m), we have \( u^T \cdot u = w^T \cdot v \). In particular, we have \( w^T \cdot (v - u) = 0 \) for all reachable states \( u, v \).

Proof. It is easily seen that if process node \( f_i \) fires \( r_i \) times, starting from a marking \( u \), we reach the following marking

\[
\begin{pmatrix}
  u_1 \\
  \vdots \\
  u_m
\end{pmatrix}
+ r_i
\begin{pmatrix}
  t_{i,1} \\
  \vdots \\
  t_{i,m}
\end{pmatrix}
\]

Thus, if all nodes \( f_1, \ldots, f_n \) fire in a firing sequence \( r_1, \ldots, r_n \) many times, respectively, we reach the marking \( u + \mathcal{I}_p \cdot r \), so that the first proposition follows. The second proposition is a simple consequence of the first where \( v = u \) holds.

Finally, if \( v \) can be reached from \( u \), then we have by the already proved first fact that \( v = u + \mathcal{I}_p \cdot r \) holds with a firing vector \( r = (r_1, \ldots, r_n) \) whose components \( r_i \) are all non-negative. Thus, we have \( w^T \cdot v = w^T \cdot (u + \mathcal{I}_p \cdot r) = w^T \cdot u + w^T \cdot \mathcal{I}_p \cdot r = w^T \cdot u + (\mathcal{I}_p \cdot w)^T \cdot r = w^T \cdot u \). Thus, if \( v \) can be reached from \( u \), we must have \( w^T \cdot (v - u) = 0 \) \( \square \)

Note that the two criteria mentioned in the above theorem are equivalent to each other, but the second criterion is more convenient since the invariants can be computed in advance so that checking the non-reachability is reduced to a simple matrix product. Note further that, by definition, the invariants \( w \) are vectors that are orthogonal to every column vector of the incidence matrix \( \mathcal{I}_p \). For Petri nets, vectors \( r \) and \( w \) with the properties \( \mathcal{I}_p \cdot r = 0 \) and \( \mathcal{I}_p^T \cdot w = 0 \) are called \( T \)- and \( S \)-invariants, respectively [89, 90, 181]. Every invariant \( w \) assigns the buffers \( z_1, \ldots, z_m \) weights \( w = (w_1, \ldots, w_m) \) such that the product \( w^T \cdot v \) remains constant under all possible firings. Every \( r \) with \( \mathcal{I}_p \cdot r = 0 \) assigns the process nodes \( f_1, \ldots, f_n \) rates \( r = (r_1, \ldots, r_n) \) so that after firing \( f_i \) \( r_i \) many times, the original state is reached again.

For example, the DPNs \( P_3, P_2, P_3, \) and \( P_4 \) have the following repetition vectors \( r_1, r_2, r_3, \) and \( r_4 \) and the following invariants \( w_1, w_2, w_3, \) and \( w_4 \), respectively (where \( p, p_1, \) and \( p_2 \) are parameters in \( \mathbb{R} \)):

- \( r_1 = (6, 3, 2)^T \cdot p \)
- \( w_1 = (1, 1, 0, 1)^T \cdot p + (1, 1, -1, 0)^T \cdot p_2 \)
- \( r_2 = (3, 2, 1)^T \cdot p \)
- \( w_2 = (1, 2, 3)^T \cdot p \)
- \( r_3 = (4, 8, 2, 1)^T \cdot p \)
- \( w_3 = (1, 1, 0, 0, 0)^T \cdot p + (0, 0, 0, 1, 1)^T \cdot p_2 \)
- \( r_4 = (0, 0, 0)^T \)
- \( w_4 = (0, 0, 0)^T \)
By the repetition vectors, we see once again that $P_1$, $P_2$, and $P_3$ have a static periodic schedule where the repetition vectors contain the number of firings of the nodes of the considered DPN.

The mapping of firing sequences of the state transition system of a DPN $P$ to associated vectors that only contain the number of firings of each node in that sequence is often called the Parikh mapping [196], and particular repetition vectors associated with a firing sequence are thus called Parikh vectors. In [150], it is proved that the set of Parikh vectors of persistent Petri nets is a lattice under the natural ordering where all components of one vector have to be less than or equal to the corresponding components of the other vector. Since deterministic DPNs have to be persistent, the same result applies to our DPNs.

In [89], several algorithms for solving the reachability problems with the above criteria of the incidence matrix are discussed. All of these algorithms aim at solving the linear equation systems with different additional constraints, e.g. whether solutions in the rational numbers exist, which is a problem with polynomial complexity, or whether solutions in the non-negative integers exist, which is NP-complete. To this end, the Farkas’ Lemma is also employed that states that exactly one of the following holds:

- there exists $r$ with $J_P \cdot r = (u - v)$ and all $r_i \geq 0$
- there exists a $y$ with $J_P^T \cdot y \geq 0$ and $(u - v) \cdot y < 0$

The above results can be used to prove the non-reachability of markings in that for given markings $u$ and $v$, we may either test whether $J_P \cdot r = v - u$ has a positive integer solution $r$ or whether $w^T \cdot (v - u) = 0$ holds for all invariants $w$ of $P$. Having this view, solving the balance equations means to check whether there is a marking $v$ that can be reached from itself. Thus, we obtain once more our result on the balance equations.

### 7.2.3 Cyclo-Static Dataflow Process Networks

Cyclo-static dataflow process networks have been introduced in [49, 96, 97] (see also [85, 199, 262]). They extend the synchronous dataflow process networks of the previous section in that the number of values produced and consumed by each node varies from one firing to the next in a cyclic pattern. Each node has thereby a number of internal states that determine different static firing rules, and these states are always traversed in the same ordering so that the nodes perform infinite loops where the internal states are periodically traversed in the same sequential order. Each node may thereby have a different number of internal states. Since each internal state defines a static firing pattern, the cyclic traversing of the internal states is observed as a periodic firing pattern.

Since these patterns are all known at compile-time, the productions and consumptions are all predictable so that the existence of bounded static peri-
odic schedules is again decidable. Note that this is not the case for the cyclo-
dynamic variant considered in [260].

\[
\begin{array}{ccc}
  x_1 & x_2 & y \\
  0 & (a :: A) & B \\
  1 & A & (b :: B)
\end{array}
\quad
\begin{array}{ccc}
  x_1 & x_2 & z & z' & y \\
  (a :: A) & B & (0 :: C) & [1] & [a] \\
  A & (b :: B) & (1 :: C) & [0] & [b]
\end{array}
\]

As a simple example, consider the firing rules of a node as shown in the left
one of the above tables. In contrast to the previous tables, the firing rules are
further distinguished by **internal states** that are marked with 0 and 1 in this
example. In each state \( n \), only the firing rules associated with this state are
allowed, and after firing, the node switches to the next state which is \((n + 1) \mod p\), if \( p \) is the number of states of a node. Instead of directly modeling
cyclo-static DPNs with nodes having internal states, we can alternatively add
a new local buffer to each node and modify the firing rules so that this local
buffer holds the internal state as shown on the right table above. For this
reason, the node specified by the above tables repeats the following behavior:
first, it takes one value from stream \( x_1 \) and sends it to the output stream
\( y \), and then it reads one value from \( x_2 \) and sends it to \( y \).

The computation of a static schedule can be similarly done as for syn-
chronous DPNs: we have to equate the number of data values produced and
consumed in each local buffer. To this end, note first that each period of the
static schedule must include a multiple of the cyclic patterns of each node.
Hence, we consider the number of cyclic patterns \( NP(f) \) that a node \( f \) has
to work through in one period of the static schedule. In detail, assume that
for a local variable \( z_i \) its producer node \( prod(z_i) \) has \( n_p \) internal states
where \( p(z_i,0), \ldots, p(z_i, n_p - 1) \) values were produced for \( z_i \) during one cyclic pattern
of \( prod(z_i) \). Analogously, assume that its consumer node \( cons(z_i) \) has \( n_c \) internal
states where \( c(z_i,0), \ldots, c(z_i, n_c - 1) \) values were consumed for \( z_i \) during
one cyclic pattern of \( cons(z_i) \). Then, the number of values produced during a
cyclic pattern of the producer node \( prod(z_i) \) is \( P(z_i) := \sum_{j=0}^{n_p-1} p(z_i, j) \) and the
number of tokens consumed during a cyclic pattern of the consumer node are
\( C(z_i) := \sum_{j=0}^{n_c-1} c(z_i, j) \). Using these numbers, we have to satisfy the following
balance equations in the long run:

\[
P(z_i) \cdot NP(prod(z_i)) = C(z_i) \cdot NP(cons(z_i))
\]

Each nontrivial solution of these balance equations yields for every node \( f \)
the numbers of periods \( NP(f) \) it will have in the period of the static schedule.
The number of its firings is then \( n_f \cdot NP(f) \) where \( n_f \) is the number of internal
states of \( f \). We therefore have proved the following theorem:

**Theorem 7.28 (Balance Equations).** Let \( n \) be a cyclo-static DPN, where \( prod(z_i) \)
and \( cons(z_i) \) denote the producer and consumer of the local variable \( z_i \in V_{loca}\).
and \( p(z_i, 0), \ldots, p(z_i, n_p - 1) \) and \( c(z_i, 0), \ldots, c(z_i, n_c - 1) \) are the number of values produced and consumed by \( \text{prod}(z_i) \) and \( \text{cons}(z_i) \), respectively, during their cyclic patterns of lengths \( n_p \) and \( n_c \), respectively. If there is a static periodic schedule where \( \text{NP}(f) \in \mathbb{N} \) is the number of cyclic patterns of node \( f \) in one period of the static schedule, then the following balance equation must hold for each local variable \( z_i \):

\[
\left( \sum_{j=0}^{n_p-1} p(z_i, j) \right) \cdot \text{NP}(\text{prod}(z_i)) = \left( \sum_{j=0}^{n_c-1} c(z_i, j) \right) \cdot \text{NP}(\text{cons}(z_i))
\]

If the resulting linear equation system should only have the trivial solution \( \text{NP}(f) = 0 \) for each node \( f \), then there is no such static periodic schedule. Otherwise, each nontrivial solution yields the numbers of cyclic patterns \( \text{NP}(f) > 0 \) node \( f \) has to work through in one period of the static schedule.

Hence, we have essentially the same benefits as for synchronous dataflow process networks even though the firing rules seem to be much less restricted.

As an example, consider the cyclo-static DPN shown in Figure 7.7. Nodes \( f \) and \( g \) have the firing rules shown in the function tables on the left and right hand sides, respectively. Hence, \( f \) has two internal states and \( g \) has three internal states that are traversed in a cyclic pattern. The dataflow graph shown
on top of Figure 7.7 is labeled with the number of data values produced and consumed during the cyclic patterns of the nodes. Thus, these are sequences of length 2 and 3, respectively. The balance equations are shown on the lower left corner of Figure 7.7 and its solutions are shown on the lower right corner.

As can be seen there is a static schedule that includes two cyclic patterns of \( f \) and one of \( g \), i.e. the firings \( f_0^0, f_1^1, f_0^0, f_1^1, g_0^0, g_1^1 \), and \( g_2^2 \) in some order, where \( f_i \) and \( g_i \) denote the firings of node \( f \) and \( g \) in state \( i \), respectively.

### 7.2.4 Boolean Dataflow Process Networks

Boolean dataflow networks (BDF)\([58–60]\) are obtained by adding two dynamic nodes to the static nodes of synchronous dataflow networks, namely the following nodes select and switch:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>x_0</td>
<td>x_1</td>
</tr>
<tr>
<td>(0 :: A)</td>
<td>(b :: B)</td>
<td>C</td>
</tr>
<tr>
<td>(1 :: A)</td>
<td>B</td>
<td>(c :: C)</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>x</td>
<td>y_0</td>
</tr>
<tr>
<td>(0 :: A)</td>
<td>(b :: B)</td>
<td>(b)</td>
</tr>
<tr>
<td>(1 :: A)</td>
<td>(b :: B)</td>
<td>(b)</td>
</tr>
</tbody>
</table>
```

It has been shown that BDF is already Turing-complete, so that checking the boundedness of such a DPN is an undecidable problem. Hence, the existence of static schedules is not guaranteed, and in general, they do not exist for these kinds of DPNs.

### 7.2.5 Tagged Dataflow Process Networks

- Arvind and Gostelow
- Sangiovanni-Vincentelli’s Tagged Tokens

### 7.3 Globally Asynchronous Locally Synchronous Systems

VAL, Sisal, Lucid, Id, …

#### 7.3.1 Synchronous Programs vs. Dataflow Programs

Synchronous programming languages are somehow related to dataflow process networks \([35, 65]\), which will be explored in this section. In particular, we consider the definition of a denotational semantics of the languages SIGNAL \([36, 37, 107, 156, 157]\) and Lustre by means of DPNs. As a denotational semantics, the processes (that are also called nodes to emphasize the relationship to DPNs) are associated with functions of type \( \text{Stream}^m_D \rightarrow \text{Stream}^n_D \). Hence, we ignore again different types and consider a universal set of values \( D \). However, to define the semantics of these languages, we need a special
value $\square$ to denote the absence of a signal, and thus we will first consider streams on $D_{\square} := D \cup \{\square\}$.

It is crucial to understand what is meant with absence of a signal: in a DPN one would simply prefer to not send a data value, and this is what will finally also be done by the synthesis procedures of SIGNAL and Lustre. However, to define their semantics, it is beneficial to introduce such a value so that certain synthesis problems can be easier described. Note also the difference between $\square$ and the value $\bot$ used in causality analysis: The value $\bot$ denotes some value (which may now also be $\square$) that is currently not yet known, but it is known that it is one of the other values. Hence, $\bot$ represents all other values and we can compute with $\bot$ whatever we can compute with all the other values instead. For this reason, we can reduce $\bot \lor 1$ by $1$ since for all boolean values $b$, we have $b \lor 1 = 1$. In contrast, $\square$ is a known value, thus it is also represented by $\bot$. Moreover, $\square$ is not comparable to values in $D$ while we have $\bot \preceq d$ for every $d \in D$ (and also $\bot \preceq \square$) for the partial order used in causality analysis.

Recall that the execution of DPNs can be controlled either by the occurrence of input data, but the need to compute output data, or by (periodic) schedules that are nothing else but clocks. As we will discuss the control of the DPNs obtained from synchronous programs by means of clocks, we have to assign to each node in such a DPN its own clock, which makes it a polychronous system. To this end, it is also beneficial to define for each stream with values $\square$ its associated clock:

**Definition 7.29 (Clock of a Stream).** For every input stream, $\sigma \in \text{Stream}_{\square\sigma}^n$, we define its clock $\text{cl}(\sigma)$ as the stream $\text{cl}(\sigma)_0, \text{cl}(\sigma)_1, \text{cl}(\sigma)_1, \ldots$ with the following elements: $\text{cl}(\sigma)_i \iff \sigma_i \neq \square$. Hence, if two streams $\sigma$ and $\sigma'$ have the same clock, then for every $i \in \mathbb{N}$ we have $\sigma_i = \square$ iff $\sigma'_i = \square$.

Since the special value $\square$ denotes the absence of a value, it does not carry information as a value does, but it is used to keep different streams aligned. Using $\square$, we may now define the process nodes as shown in Figure 7.8 that essentially define the language SIGNAL [36, 37, 107, 156, 157] and Lustre [118–120] in terms of DPNs.

As can be seen, all of these nodes are static nodes that consume by each firing one data value of each input buffer and generate one value on their output buffers (except for the initial behavior of $\text{pre}$ and the local buffer $z$ of $\$\$). The $\text{pre}$ operator simply adds the additional value $\square$ at the beginning of a stream and therefore shifts all components of the stream one step to the right. This has the consequence that $x$ and $\text{pre}(x)$ do not have the same clock. For this reason, a similar operator $\$\$ is defined as follows: for some constant $c$ and a stream $x$, the stream $x \$ c$ is obtained by adding $c$ as first element of $x$, but keeping the clock of $x$ for $x \$ c$. Given input streams $x = x_0, x_1, \ldots$ and $y = y_0, y_1, \ldots$, the stream $x \text{ fby} y$ (pronounced ‘followed by’) is simply $x_0, y_1, y_2, \ldots$. Sometimes, also the operator $\text{cby}$ (pronounced ‘continued by’) is used which can be defined as follows: $x \text{ cby} y := x \text{ fby} y(x)$. Hence, this operator concatenates the first value of $x$ to the left of the entire stream $y$ so that $x_0,$
### 7.3 Globally Asynchronous Locally Synchronous Systems

<table>
<thead>
<tr>
<th>pre</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$$z$</td>
</tr>
<tr>
<td>$A$</td>
<td>$(\boxempty :: A) B$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$(a :: A) B$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$a \neq \boxempty$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(b :: B)$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$a \neq \boxempty$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(b :: B)$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A) (\boxempty :: B)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cby</th>
<th>fby</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ $y$ $x$ cby $y$</td>
<td>$x$ $y$ $x$ fby $y$</td>
</tr>
<tr>
<td>$(a :: A) B$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(b :: B)$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$cby$</td>
<td>$fby$</td>
</tr>
<tr>
<td>$x$ $y$ $x$ cby $y$</td>
<td>$x$ $y$ $x$ fby $y$</td>
</tr>
<tr>
<td>$(a :: A) B$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(b :: B)$</td>
<td>$(a :: A) (b :: B)$</td>
</tr>
<tr>
<td>$cby$</td>
<td>$fby$</td>
</tr>
<tr>
<td>default</td>
<td>when</td>
</tr>
<tr>
<td>$x$ $y$ $x$ default $y$</td>
<td>$x$ $b$ $x$ when $b$</td>
</tr>
<tr>
<td>$(a :: A) (b :: B)$</td>
<td>$(a :: A) (1 :: B)$</td>
</tr>
<tr>
<td>$(a :: A) (b :: B)$</td>
<td>$(a :: A) (0 :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$\boxempty$</td>
</tr>
<tr>
<td>$(a :: A) (\boxempty :: B)$</td>
<td>$(a :: A) (\boxempty :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A) (\boxempty :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A) (\boxempty :: B)$</td>
</tr>
<tr>
<td>$\boxempty$</td>
<td>$(a :: A)$</td>
</tr>
<tr>
<td>$(\boxempty :: A)$</td>
<td>$(a :: A) (\boxempty :: B)$</td>
</tr>
</tbody>
</table>

**Fig. 7.8.** Operators of SIGNAL as DPN Process Nodes

$y_0, y_1, y_2, \ldots$ is obtained. The differences between these operators are best understood by considering the following example:

<table>
<thead>
<tr>
<th>$x$</th>
<th>1 0 0 0 4 4 6 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x $ 0</td>
<td>0 0 0 0 1 4 6 6 ...</td>
</tr>
<tr>
<td>0 cby $x$</td>
<td>0 0 0 0 4 4 6 6 ...</td>
</tr>
<tr>
<td>0 fby $x$</td>
<td>0 0 0 0 4 4 6 7 ...</td>
</tr>
</tbody>
</table>

The when operator forwards the data value read from input $x$ to its output whenever the data value read at the same time from input $b$ is true. Otherwise, the value of $x$ is consumed from the input buffer, but $\boxempty$ is forwarded to the output.

Finally, the default operator first reads values $a$ and $b$ from its input buffers $x$ and $y$, and then checks whether $a = \boxempty$ holds. If this should be the case, the value $b$ is forwarded to the output (regardless whether it is $\boxempty$ or not); otherwise, the value $a$ is forwarded to the output buffer.

As can be seen, there are two operators that may introduce $\boxempty$: the pre operator prefixes its argument stream by a single value $\boxempty$, and the when operator replaces elements $x_i$ by $\boxempty$ whenever $b_i \neq 1$ holds. Moreover, only the default operator is able to remove $\boxempty$ in its first input stream $x$ by values different to $\boxempty$ of its second input stream $y$. Hence, default and when modify the clocks of their input streams. To this end, we make the following definition:
Definition 7.30 (Clock Lattice). Any stream \( \sigma \in \text{Stream}_B \) is called a clock, and the set of clocks is partially ordered by the pointwise implication, i.e., we define \( \sigma_1 \preceq \sigma_2 \) if \( \sigma_1 \rightarrow \sigma_2 \) holds. Since \( \sup(\{\sigma_1, \sigma_2\}) = \sigma_1 \lor \sigma_2 \) and \( \inf(\{\sigma_1, \sigma_2\}) = \sigma_1 \land \sigma_2 \) holds, the set of clocks is a lattice.

We also say that \( \sigma_1 \) is slower than \( \sigma_2 \) if \( \sigma_1 \preceq \sigma_2 \) holds, since each time \( \sigma_1 \) ticks, there is also a tick of \( \sigma_2 \).

It remains to define the semantics of typical arithmetic and boolean operators. We will write \( \oplus \) in the following to denote any such binary operator (and the discussion can be made analogously for operators having more arguments). Clearly, these operations should be applied on the corresponding components of given input streams to generate an output stream. We have no problem to define the output in the case where both input streams start with a value different to \( \Box \). In the case where both streams start with the value \( \Box \), it is also quite natural to produce the value \( \Box \). Hence, the only questionable cases that remain are those where from one of the arguments the value \( \Box \) is read while from the other one a value different to \( \Box \) is read. We therefore have to complete the following firing rules of such a function \( \oplus \):

\[
\begin{array}{ccc}
x & \oplus & y \rightarrow x \oplus y \\
(\Box :: A) & (\Box :: B) & 1 \rightarrow [\Box] \\
(\Box :: A) & (b :: B) & b \neq \Box \rightarrow ? \\
(a :: A) & (\Box :: B) & a \neq \Box \rightarrow ? \\
(a :: A) & (b :: B) & a \neq \Box \land b \neq \Box \rightarrow [a \oplus b]
\end{array}
\]

The questionable cases are the second and third firing rules. In principle, we have here three different choices to deal with these two cases:

- First, we may simply forbid these two cases, which means that the argument streams of nodes \( \oplus \) must have the same clock. Such DPNs are called clock-consistent DPNs. DPN that are not clock-consistent would then be not well-defined and would therefore be discarded.
- Second, we could define \( x \oplus y = \Box \) if \( x = \Box \) or \( y = \Box \) should hold, thus we would complete the output of the questionable firing rules in the above table by producing output \( [\Box] \). This allows us to consider even DPNs that were not clock-consistent.
- Third, we might ignore occurrences of \( \Box \) and might wait instead for the next element in the stream that is different to \( \Box \) for the computation of the desired result value. In this case, the output of the questioned cases would be \( [\ ] \).

There are good reasons against the latter two choices, so that we will prefer the restriction to clock-consistent DPNs: If we would choose the second choice, the input streams would be desynchronized, and the ‘wrong’ components of the input streams could be used as arguments. ‘Wrong’ means here that we want to remove the values \( \Box \) later on from the streams and wish that the
results remain the same when $\square$ are also removed from the output streams. Thus, we cannot choose alternative two.

If we would choose the third alternative, even very simple dataflow programs could require infinite memory requirements as will be shown by the following SIGNAL program with one integer input $x$, one integer output $y$ and two local variables $z_1$ and $z_2$:

$$
\begin{cases}
  z_1 = 0 	ext{ cby } \neg z_1 \\
  z_2 = x \text{ when } z_1 \\
  y = x + z_2
\end{cases}
$$

According to our denotational semantics, it is not difficult to see that $z_1$ denotes the stream $0, 1, 0, 1, \ldots$, and that $z_2$ therefore denotes the stream $\square, x_1, \square, x_3, \square, x_5, \ldots$. The meaning of $y$ depends on the three choices mentioned above to define the meaning of $+$:

- If we demand that $x$ and $z_2$ must have the same clock, we have to demand that at least the values with even indices $x_0, x_2, \ldots$ are equal to $\square$. Assuming this, and that the other values are different to $\square$, the value of $y$ would be $\square, x_1 + x_1, \square, x_3 + x_3, \square, x_5 + x_5, \ldots$.
- If we would define $x \oplus y = \square$ if $x = \square$ or $y = \square$ should hold, then we would obtain the same result as above.
- Finally, if we would ignore occurrences of $\square$ and would wait instead for the next element in the stream that is different to $\square$ for the computation of the desired result value, then we would obtain the following stream (if $x_i \neq \square$ holds for all $i$): $x_0 + x_1, x_1 + x_3, x_2 + x_5, \ldots$ whose elements are $x_i + x_{2i+1}$.

Hence, the third alternative has the consequence that the above program requires unbounded memory for its execution since it has to store the elements $x_i, \ldots, x_{2i+1}$ for its computations which is a growing number of elements.

For this reason, we demand that input streams of boolean and arithmetic operators must have the same clock, i.e., we demand the following clock consistency condition:

**Definition 7.31 (Clock Consistency).** We define the clock consistency condition of a SIGNAL expression as follows:

- $cc(x) :\iff 1$ for variables $x$
- $cc(c) :\iff 1$ for constants $c$
- $cc(pre(x)) :\iff cc(x)$
- $cc(x \& c) :\iff cc(x)$
- $cc(x \&by\ y) :\iff cc(x) \land cc(y)$
- $cc(x \& cby\ y) :\iff cc(x) \land cc(y)$
- $cc(x \& default\ y) :\iff cc(x) \land cc(y)$
- $cc(x \& when\ b) :\iff cc(x) \land cc(b)$
- $cc(x \& y) :\iff (cl(x) \leftrightarrow cl(y)) \land cc(x) \land cc(y)$
For a program $P$ with equations $\ell_1 = r_1, \ldots, \ell_n = r_n$ we moreover define
\[
cc(P) :\iff \bigwedge_{i=1}^n (\text{cl}(\ell_i) \leftrightarrow \text{cl}(r_i)) \land cc(\ell_i) \land cc(r_i)
\]
The only crucial case for the clock consistency of expressions is the clock consistency of expressions $x \otimes y$ where we have to demand that $x$ and $y$ have the same clock (note that $\text{cl}(x) \leftrightarrow \text{cl}(y)$ means that the clocks $\text{cl}(x)$ and $\text{cl}(y)$ are the same for all points of time, since $\leftrightarrow$ is meant to be applied componentwise as any other boolean or arithmetic operator). In all other cases, we simply traverse the syntax tree of the expression to find deeper nested subexpressions of the form $x \otimes y$ to include their clock consistency expression in a conjunction.

Similarly, the clock consistency of a program mainly demands that the clocks of the left and right hand sides of the equations are the same, and that both the left and right hand side expressions are clock-consistent expressions. The clock of a SIGNAL expression can moreover be computed as follows:

**Lemma 7.32.** The clocks of derived streams can be recursively computed as follows, where the boolean operations have to be applied componentwise on the streams and constants are to be read as streams consisting of this constant value only:

- $\text{cl}((\text{pre}(x)) = 0 \ cby \ \text{cl}(x)$
- $\text{cl}(x \& c) = \text{cl}(x)$
- $\text{cl}(x_1 \ fby \ x_2) = \text{cl}(x_1) \ fby \ \text{cl}(x_2)$
- $\text{cl}(x_1 \ cby \ x_2) = \text{cl}(x_1) \ cby \ \text{cl}(x_2)$
- $\text{cl}(x_1 \ default \ x_2) = \text{cl}(x_1) \lor \text{cl}(x_2)$
- $\text{cl}(x_1 \ when \ x_2) = \text{cl}(x_1) \land (x_2 = 1)$
- $\text{cl}(x_1 \& x_2) = \text{cl}(x_1) \ (or \ \text{cl}(x_1 \& x_2) = \text{cl}(x_2) \ since \ we \ demand \ \text{cl}(x_1) = \text{cl}(x_2))$

**Proof.** Note that $x_2 = 1$ is also a boolean stream, but slower than $\text{cl}(x_2)$, since the latter is true whenever $x_2 \neq \Box$, whereas the former is only true when $x_2$ is 1 (but false when $x_2$ is 0). The equations can be easily proved by considering the different cases of the firing rules that define the operators, e.g. for $x_1 \ default \ x_2$ and $x_1 \ when \ x_2$, the proofs are given by the following tables that list only the first elements of the streams mentioned in the headers of the tables:
For example, we can now compute the clock consistency of the previous dataflow program:

\[
P := \begin{cases} 
  z_1 = 0 \text{ cby } \neg z_1 \\
  z_2 = x \text{ when } z_1 \\
  y = x + z_2
\end{cases} \quad \text{cc}(P) \iff \left( (cl(z_1) \leftrightarrow 1 \text{ cby } cl(z_1)) \land (cl(z_2) \leftrightarrow cl(x) \land (z_1 = 1)) \land (cl(y) \leftrightarrow cl(x)) \land (cl(y) \leftrightarrow cl(z_2)) \right)
\]

From \(cl(z_1) \leftrightarrow 1 \text{ cby } cl(z_1)\), it follows that \(cl(z_1) \leftrightarrow 1\). Recalling that \(z_1 = 0, 1, 0, 1, \ldots\), we conclude that \(cl(z_2)\) is false at all even points of time and is \(cl(x)\) at the odd points of time. Since \(cl(y)\), \(cl(x)\), and \(cl(z_2)\) must all be the same, these clocks must also be false at the even points of time, and must be \(cl(x)\) at the odd points of time. Hence, \(cl(x)\) must be false at all even points of time, and may be arbitrary at the odd points of time to make the program clock-consistent.

In [36, 37, 107, 156], a clock calculus has been defined that does not derive a clock consistency condition as defined above, but instead encodes streams that may contain \(\Box\) in terms of arithmetic streams. The encoding is different for boolean and non-boolean streams, and is defined as follows:

**Definition 7.33 (Clock Calculus Encoding).** For any non-boolean value \(x \in D\Box\) and any boolean value \(b \in D\Box\), we define their clock calculus encodings \(\epsilon(x)\) and \(\epsilon(b)\) as follows:
\(\epsilon(x) := \begin{cases} 0 & \text{if } x = \square \\ 1 & \text{otherwise} \end{cases} \quad \epsilon(b) := \begin{cases} -1 & \text{if } b = 0 \\ 0 & \text{if } b = \square \\ 1 & \text{if } b = 1 \end{cases} \)

\(\epsilon(x)\) is therefore essentially the clock of \(x\) (except for using the numbers 0 and 1 instead of the boolean constants 0 and 1), while for boolean values, \(\epsilon(b)\) encodes not only the presence of data, but also the boolean values themselves including \(\square\). We abandoned the definition of function evaluations where one of the arguments is \(\square\) while another one is different to \(\square\). For the following lemma, we define in these cases \(\square\), which is necessary for the validity of the lemma, but irrelevant for the later analysis, since we anyway demand the clock-consistency of the DPN.

**Lemma 7.34 (Clock Calculus [36, 37, 107, 156])**. The following equations are valid for the encoding given in Definition 7.33 if the arithmetic operations for boolean and non-boolean streams are computed modulo 3 and 2, respectively (where the results modulo 3 should belong to \((-1, 0, 1)\):

- \(\epsilon(-b) = -\epsilon(b)\)
- \(\epsilon(b_1 \land b_2) = \epsilon(b_1)\epsilon(b_2) (\epsilon(b_1)\epsilon(b_2) - (1 + \epsilon(b_1) + \epsilon(b_2)))\)
- \(\epsilon(b_1 \lor b_2) = \epsilon(b_1)\epsilon(b_2) (1 - (\epsilon(b_1) + \epsilon(b_2) + \epsilon(b_1)\epsilon(b_2)))\)
- \(\epsilon(b_1 \text{ when } b_2) = -\epsilon(b_1) \cdot \epsilon(b_2)(1 + \epsilon(b_2))\)
- \(\epsilon(b_1 \text{ default } b_2) = \epsilon(b_1) + \epsilon(b_2) - \epsilon(b_1)^2\epsilon(b_2)\)

Moreover, the equivalence \(cl(b_1) \leftrightarrow cl(b_2)\) for streams \(b_1\) and \(b_2\) can be expressed as \(\epsilon(b_1)^2 = \epsilon(b_2)^2\).

**Proof.** The proofs are easily obtained by evaluating the equations for the possible finitely many cases that can occur for \(\epsilon(b_1), \epsilon(b_2), \epsilon(x_1),\) and \(\epsilon(x_2)\). Note that we temporarily defined \(\square \land b = b \land \square = \square\) and similarly for \(\neg\) and \(\lor\). \(\square\)

Different computations for \(\epsilon(b_1 \text{ when } b_2)\) and \(\epsilon(b_1 \text{ default } b_2)\) are mentioned in [36, 37, 107, 156] for boolean and non-boolean streams. However, we remark that the above equations that hold for boolean streams do also hold for non-boolean ones, since values obtained for non-booleans \(\epsilon(x) \in \{0, 1\}\) are a subset of the values that may be obtained for a booleans \(\epsilon(b) \in \{-1, 0, 1\}\). It is however beneficial to make use of some simplifications after the following definition:

**Definition 7.35.** For any DPN \(P\) with equations of the form listed below where each right hand side contains exactly one operator, we generate an equation system according to the following rules, where \(b, b_1,\) and \(b_2\) are booleans, and \(x,\) \(x_1,\) and \(x_2\) are non-booleans:

- \(\epsilon(y = -b) := \{\epsilon(y) = \epsilon(-b)\}\)
- \(\epsilon(y = b_1 \land b_2) := \{\epsilon(y) = \epsilon(b_1 \land b_2), \epsilon(b_1)^2 = \epsilon(b_2)^2\}\)
- \(\epsilon(y = b_1 \lor b_2) := \{\epsilon(y) = \epsilon(b_1 \lor b_2), \epsilon(b_1)^2 = \epsilon(b_2)^2\}\)
- \(\epsilon(y = b \& c) := \{\epsilon(y)^2 = \epsilon(b)^2\}\)
P = \bigcup_{i=1}^n \epsilon_i = r_i.

The idea is to demand for boolean equations \( y = r \) that \( \epsilon(y) = \epsilon(r) \) holds, i.e., that the streams have exactly the same elements, and to additionally demand for binary boolean operators that their arguments have the same clocks which is specified by the equation \( \epsilon(b_1)^2 = \epsilon(b_2)^2 \). For non-boolean equations \( y = r \) we demand only that the two streams have the same clock by the equation \( \epsilon(y)^2 = \epsilon(r)^2 \), and additionally demand for \( y = x_1 \oplus x_2 \) that the streams \( x_1 \) and \( x_2 \) are clock-consistent by the equation \( \epsilon(x_1)^2 = \epsilon(x_2)^2 \).

**Why not simply generating \( \epsilon(y) = \epsilon(r) \) for non-boolens also?**

There are a few exceptions from this general idea: for \( \epsilon(y = b \% c) \) we can only demand that the \( y \) and \( b \% c \) are clock consistent, since the values of \( b \) were shifted to the right to the places where \( b \) is not \( \Box \). Moreover, for \( \epsilon(y = x_1 \text{ when } b_2) \) and \( \epsilon(y = x_1 \text{ default } x_2) \) we made some simplifications: we could equivalently generate the equations \( \epsilon(y)^2 = \epsilon(x_1 \text{ when } b_2)^2 \) and \( \epsilon(y)^2 = \epsilon(x_1 \text{ default } x_2)^2 \), respectively, but used the equations mentioned in the above definition to reduce the degree of the polynomials. For this reason, it is possible to generate an equation system with polynomials of maximal degree 2.

**Theorem 7.36.** For any SIGNAL program \( P \), the equation system \( \epsilon(P) \) has a solution iff \( P \) is clock consistent.

Hence, it is easily seen that our definition of clock consistency matches with the one given in [36, 37, 107, 156]. For this reason, the clock consistency problem can be represented in terms of arithmetic equation systems as well.

**for periodic scheduling of cyclo-static processes, see [51, 52]***

### 7.3.2 Endochrony and Isochrony

In the previous section, we have considered the syntax and semantics of the synchronous dataflow languages SIGNAL and Lustre. Clearly, we can implement clock-consistent programs of these languages in terms of DPNs by directly following the semantics given in the previous section. In this case, the nodes can all be fired with the same rates, where in each reaction all inputs are read and all outputs are written, so that buffers only need to store one value.
A natural question to optimize the efficiency of these DPNs is to remove the communication overhead given by the explicit use of the value $\Box$ in the streams. Clearly, some important information is then lost: Recall that the appearances of the values $\Box$ synchronize the values in the streams of the different variables so that all values referring to a reaction appear at the same position; e.g. for the behaviors $\xi_1$, $\xi_2$, and $\xi_3$ shown in Figure 7.13 all values that are written in one column belong to the same reaction of the system. Using $\Box$, it is furthermore possible that even though a reaction does not need a ‘real’ value from a particular input or does not write a ‘real’ value to an output, since in these cases, we simply demand that $\Box$ is read or written.

Hence, if no values $\Box$ were generated (and therefore also not read), the mapping of values to reactions is lost in the behaviors. In the following, we therefore talk about synchronous and asynchronous behaviors depending on whether values $\Box$ are used. Using asynchronous behaviors to avoid the communication overhead introduced by $\Box$, it is therefore necessary for a synchronous system to resynchronize the input streams so that the values to be used in a synchronous reaction are correctly determined. Systems having this property are called endochronous systems, where the greek words ‘endo’ and ‘chronos’ indicate that the timing comes from within the system/data streams and is not given explicitly by means of a clock signal or by means of $\Box$ values.

\[
\begin{array}{c|ccc|c}
  & x_1 & x_2 & x_3 & y \\
 1 & A & b & B & c \\
 0 & A & b & B & c \\
\end{array}
\]

$\xi(x_1) : 1 \ 0 \ 0 \ 1 \ 1 \ldots$

$\xi(x_2) : 1 \ 3 \ 5 \ 7 \ 9 \ldots$

$\xi(x_3) : 0 \ 2 \ 4 \ 6 \ 8 \ldots$

$\xi(y) : 1 \ 2 \ 4 \ 7 \ 9 \ldots$

Fig. 7.9. A synchronous if-then-else node (left) with a possible behavior (right).

Before presenting a formal foundation of endochronous systems, we consider some motivating examples. The firing rules of a simple synchronous\(^4\) if-then-else node could be initially given by the firing rules shown on the left of Figure 7.9. Clearly, the node reads values $a$, $b$, and $c$ from the inputs $x_1$, $x_2$ and $x_3$ and copies either $b$ or $c$ to the output depending on whether $a$ is true or false as can be seen with the example behavior on the right of Figure 7.9.

The if-then-else node shown in Figure 7.9 consumes values of both $x_2$ and $x_3$ and only forwards one of these values, while the other one is lost. One might now have the wish to implement a deterministic merge node where depending on the value read from $x_1$ either a value of $x_2$ or $x_3$ is read and copied to the output. It is not difficult to implement such a node by an asynchronous node as shown in Figure 7.10.

\(^4\) It is called synchronous, since it reads in each reaction one value from all inputs and writes exactly one value to all of its outputs.
7.3 Globally Asynchronous Locally Synchronous Systems 261

\[
x_1 \quad x_2 \quad x_3 \quad y
\]

\begin{align*}
(1 \downarrow A) & \; (b \uparrow B) \; C \; [b] \\
(0 \downarrow A) & \; (\square \uparrow B) \; (c \uparrow C) \; [c]
\end{align*}

\[
\xi(x_1) : 1 \; 0 \; 0 \; 1 \; 1 \ldots \\
\xi(x_2) : 1 \; 0 \; 2 \; 1 \; 3 \ldots \\
\xi(x_3) : 1 \; 3 \; 5 \; 7 \; 9 \ldots \\
\xi(y) : 1 \; 0 \; 2 \; 3 \; 5 \ldots
\]

Fig. 7.10. An asynchronous deterministic merge node (left) with a possible behavior (right).

\[
x_1 \quad x_2 \quad x_3 \quad y
\]

\begin{align*}
(1 \downarrow A) & \; (b \uparrow B) \; (\square \uparrow C) \; [b] \\
(0 \downarrow A) & \; (\square \uparrow B) \; (c \uparrow C) \; [c]
\end{align*}

\[
\xi(x_1) : 1 \; 0 \; 0 \; 1 \; 1 \ldots \\
\xi(x_2) : 1 \; \square \; \square \; 3 \; 5 \ldots \\
\xi(x_3) : \square \; 0 \; 2 \; \square \; \square \ldots \\
\xi(y) : 1 \; 0 \; 2 \; 3 \; 5 \ldots
\]

Fig. 7.11. A synchronous deterministic merge node (left) with a possible behavior (right).

A synchronous implementation of that node has to introduce \( \square \) in the firing rules whenever no 'real' value is read from that input stream, which is shown in Figure 7.11. Note that the introduced values \( \square \) synchronize the input streams so that in the behavior shown on the right of Figure 7.11 values belonging to one reaction are listed in single columns. Clearly, the synchronous merge node shown in Figure 7.11 can not react to all input streams, but it is easily seen that for each behavior of the corresponding asynchronous system of Figure 7.10 there is a uniquely defined behavior of the synchronous one shown in Figure 7.11.

We wish now to implement synchronous systems like the one shown in Figure 7.11 without using \( \square \) for synchronizing the inputs, i.e., we wish to implement synchronous systems by asynchronous DPNs. Some synchronous systems allow us such an asynchronous implementation while others do not. The above synchronous deterministic merge can be easily implemented asynchronously as shown in Figure 7.10.

There are synchronous systems that can uniquely resynchronize their inputs while other synchronous system do not have this ability. The sequential-OR and parallel-OR nodes shown in Figure 7.12 are used to demonstrate this with two behaviors \( \xi_1 \in \text{Bhv}_{\text{sequential-OR}} \) and \( \xi_1, \xi_2 \in \text{Bhv}_{\text{parallel-OR}} \).

If we remove the values \( \square \) from the input streams of a behavior of the sequential-OR, it is no problem to resynchronize these input streams afterwards, since the only firing rule that makes use of \( \square \) does not produce a 'real' value, and we can simply omit reactions where all variables have value \( \square \) (these are called silent actions). However, for the parallel-OR, this is not possible, since the two behaviors \( \xi_1 \) and \( \xi_2 \) listed in Figure 7.12 become the same for the input variables \( x_1 \) and \( x_2 \), but the output stream is different for these behaviors. Hence, it is not possible to uniquely resynchronize the in-
put streams for parallel-OR, while it is easily possible for the sequential-OR. Hence, sequential-OR is endochronous, while parallel-OR is not.

The example with the sequential-OR and the parallel-OR already demonstrates that we have no problem to resynchronize input streams of nodes where all firing rules either require ‘real’ values from all inputs or perform a silent action (where only values $\Box$ are read from all inputs and values $\Box$ are written to all outputs). There might be problems when firing rules use patterns where both $\Box$ and other values are used.

Based on the above considerations, we next formally define endochronous systems, i.e., systems that have the ability to resynchronize their input streams that have been desynchronized by the elimination of $\Box$ values. Endochronous systems have been originally introduced in [27, 30, 32, 33], but we follow the alternative definitions given in [157, 245, 246]. To this end, we first have to introduce some auxiliary definitions in advance:

**Definition 7.37 (Stretching Function).** A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is called a stretching function, if the following holds:

- $\forall t \in \mathbb{N}. t \leq f(t)$
- $\forall t_1, t_2 \in \mathbb{N}. t_1 \leq t_2 \rightarrow f(t_1) \leq f(t_2)$

We write $\text{Stretch}(f)$ if $f$ satisfies the above requirements.

The intuition of a stretching function is that points of time are mapped to later points of time. We will use stretching functions to define the following relations between variable assignments:

**Definition 7.38 (Stretch and Flow Orders and Equivalences).** For variable assignments $\xi$ and $\rho$, we define the following relations: is a stretching of a variable assignment $\rho$, written as $\rho \prec_{cl} \xi$ if there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that the following hold:
For reasons given below, relations \(\rho \prec_{\text{cl}} \xi \) and \(\rho \prec_{\text{fl}} \xi\) are called stretch and flow orders, and \(\approx_{\text{cl}}\) and \(\approx_{\text{fl}}\) are called stretch and flow equivalences, respectively.

Intuitively, \(\rho \prec_{\text{cl}} \xi\) means that \(\xi\) is obtained from \(\rho\) by inserting points of times where all variables are mapped to \(\square\), i.e., by inserting silent actions in \(\rho\). Note that we therefore have \(\{t \in \mathbb{N} \mid \exists x \in \mathcal{V}. \|x\|_\xi(f(t)) \neq \square\} = f(\{t \in \mathbb{N} \mid \exists x \in \mathcal{V}. \|x\|_\rho(f(t)) \neq \square\})\). In contrast, \(\rho \prec_{\text{fl}} \xi\) means that the stretching is not done reaction-wise, but variable-wise, i.e., streams assigned to different variables are stretched by possibly different functions, which is just achieved by exchanging the ordering of the quantifiers \(\exists f. \forall x \in \mathcal{V} \) to \(\forall x \in \mathcal{V}. \exists f.\). The relations \(\rho \approx_{\text{cl}} \xi\) and \(\rho \approx_{\text{fl}} \xi\) are analogously defined for the order relations \(\rho \prec_{\text{cl}} \xi\) and \(\rho \prec_{\text{fl}} \xi\). Due to these observations, we make the following observations:

**Lemma 7.39 (Flow Relations refine Stretch Relations).** For all variable assignments \(\rho\) and \(\xi\) over variables \(\mathcal{V}\), we have the following implications:

- \(\rho \prec_{\text{cl}} \xi\) implies \(\rho \prec_{\text{fl}} \xi\)
- \(\rho \approx_{\text{cl}} \xi\) implies \(\rho \approx_{\text{fl}} \xi\)
- \(\rho \prec_{\text{fl}} \xi\) implies for every \(x \in \mathcal{V}\) that \(\rho|_x \prec_{\text{cl}} \xi|_x\)
- \(\rho \approx_{\text{fl}} \xi\) implies for every \(x \in \mathcal{V}\) that \(\rho|_x \approx_{\text{cl}} \xi|_x\)
For example, in Figure 7.13, we have $\rho \prec_{cl} \xi$ and $\rho \prec_{cl} \rho$, but neither $\rho \prec_{cl} \eta$ nor $\xi \prec_{cl} \rho$. We also have $\rho \prec_{fl} \xi$, $\rho \prec_{fl} \rho$, and $\rho \prec_{fl} \eta$, but neither $\xi \prec_{fl} \eta$ nor $\xi \prec_{fl} \xi$. However, all variables assignments are flow equivalent ($\approx$), and $\rho$, $\rho$, and $\xi$ are stretch equivalent.

We already used the names stretch and flow orders and equivalences, and this is formally justified by the next lemmas:

**Lemma 7.40 (Stretch and Flow Relations are Partial Orders).** The relations $\rho \prec_{cl} \xi$ and $\rho \prec_{fl} \xi$ given in Definition 7.38 for variable assignments $\rho$ and $\xi$ are partial order relations, i.e.,

- $\forall \xi, \xi \prec_{cl} \xi$
- $\forall \xi_1, \xi_2, \xi_1 \prec_{cl} \xi_2 \land \xi_2 \prec_{cl} \xi_1 \Rightarrow \xi_1 = \xi_2$
- $\forall \xi_1, \xi_2, \xi_3, \xi_1 \prec_{cl} \xi_2 \land \xi_2 \prec_{cl} \xi_3 \Rightarrow \xi_1 \prec_{cl} \xi_3$

and analogously

- $\forall \xi, \xi \prec_{fl} \xi$
- $\forall \xi_1, \xi_2, \xi_1 \prec_{fl} \xi_2 \land \xi_2 \prec_{fl} \xi_1 \Rightarrow \xi_1 = \xi_2$
- $\forall \xi_1, \xi_2, \xi_3, \xi_1 \prec_{fl} \xi_2 \land \xi_2 \prec_{fl} \xi_3 \Rightarrow \xi_1 \prec_{fl} \xi_3$

The proofs are not too difficult: Reflexivity is easily seen by using the stretching function $f(t) := t$, and transitivity is obtained by using the composition of the functions used to define $\xi_1 \prec_{cl} \xi_2$ and $\xi_2 \prec_{cl} \xi_1$. Antisymmetry is a bit more difficult to prove formally, but it is intuitively easily seen by recalling that $\rho \prec_{cl} \xi$ means that $\xi$ is obtained from $\rho$ by inserting ‘silent actions’ in $\rho$, i.e., points of time where all variables are mapped to $\Box$. If this can be done for $\rho$ to obtain $\xi$ and vice versa, it means that $\rho = \xi$ must hold. The arguments for flow ordering are analogous.

We moreover have the following confluence property of these partial orders:

**Lemma 7.41 (Confluence of Stretch and Flow Orders).** Stretch and flow ordering satisfy the following confluence properties:

- For all variable assignments $\xi, \rho_1, \rho_2$ with $\rho_1 \prec_{cl} \xi$ and $\rho_2 \prec_{cl} \xi$, there is a variable assignment $\xi'$ with $\xi' \prec_{cl} \rho_1$ and $\xi' \prec_{cl} \rho_2$.
- For all variable assignments $\xi, \rho_1, \rho_2$ with $\xi \prec_{fl} \rho_1$ and $\xi \prec_{fl} \rho_2$, there is a variable assignment $\xi'$ with $\rho_1 \prec_{fl} \xi'$ and $\rho_2 \prec_{fl} \xi'$.
- For all variable assignments $\xi, \rho_1, \rho_2$ with $\rho_1 \prec_{fl} \xi$ and $\rho_2 \prec_{fl} \xi$, there is a variable assignment $\xi'$ with $\rho_1 \prec_{fl} \xi'$ and $\rho_2 \prec_{fl} \xi'$.
- For all variable assignments $\xi, \rho_1, \rho_2$ with $\xi \prec_{fl} \rho_1$ and $\xi \prec_{fl} \rho_2$, there is a variable assignment $\xi'$ with $\rho_1 \prec_{fl} \xi'$ and $\rho_2 \prec_{fl} \xi'$.

The above lemma is easy to see, once you recall that $\rho \prec_{cl} \xi$ means that $\xi$ is obtained from $\rho$ by inserting silent actions, or vice versa, $\rho$ is obtained from $\xi$ by removing some silent actions. Hence, if $\rho_1 \prec_{cl} \xi$ and $\rho_2 \prec_{cl} \xi$ holds, one can remove from $\rho_2$ those silent actions that have already been removed in
\(\rho_1\), or vice versa remove the silent actions from \(\rho_1\) that have been already removed in \(\rho_2\). The analogous arguments hold for the flow ordering, and when ‘removing’ is replaced with ‘inserting’, also the other confluences are proved.

The above lemma has the following immediate consequence:

**Lemma 7.42 (Stretch and Flow Equivalence Relations).** The relations \(\rho \approx_{\text{cl}} \xi\) and \(\rho \approx_{\text{fl}} \xi\) given in Definition 7.38 for variable assignments \(\rho\) and \(\xi\) are equivalence relations, i.e.,

- \(\forall \xi, \xi \approx_{\text{cl}} \xi\)
- \(\forall \xi_1, \xi_2, \xi_1 \approx_{\text{cl}} \xi_2 \rightarrow \xi_2 \approx_{\text{cl}} \xi_1\)
- \(\forall \xi_1, \xi_2, \xi_3, \xi_1 \approx_{\text{cl}} \xi_2 \wedge \xi_2 \approx_{\text{cl}} \xi_3 \rightarrow \xi_1 \approx_{\text{cl}} \xi_3\)

and analogously

- \(\forall \xi, \xi \approx_{\text{fl}} \xi\)
- \(\forall \xi_1, \xi_2, \xi_1 \approx_{\text{fl}} \xi_2 \rightarrow \xi_2 \approx_{\text{fl}} \xi_1\)
- \(\forall \xi_1, \xi_2, \xi_3, \xi_1 \approx_{\text{fl}} \xi_2 \wedge \xi_2 \approx_{\text{fl}} \xi_3 \rightarrow \xi_1 \approx_{\text{fl}} \xi_3\)

Again, the proofs are immediate, and for proving transitivity, we need the above confluence lemma for the partial orders \(\prec_{\text{el}}\) and \(\prec_{\text{fl}}\). It is also interesting to note that for each of the equivalence classes defined by \(\approx_{\text{cl}}\) and \(\approx_{\text{fl}}\), we can easily determine a uniquely defined representative since all stretch equivalent variable assignments yield the same variable assignment when all silent actions are removed from them, and all flow equivalent variable assignments yield the same variable assignment when all values \(\Box\) are removed from the streams assigned to single variables.

An important lemma that allows one to derive stretch and flow equivalence for subsets of variables and in case of flow equivalence also in the reverse direction, is the following one:

**Lemma 7.43 (Projection Lemma).** Given variable assignments \(\xi\) and \(\rho\) over the variables \(\mathcal{W} := \mathcal{U} \cup \mathcal{V}\), the following holds:

- \(\xi \approx_{\text{cl}} \rho\) implies \(\xi|_U \approx_{\text{cl}} \rho|_U\) and \(\xi|_V \approx_{\text{cl}} \rho|_V\), but the converse is, in general, false.
- \(\xi \approx_{\text{fl}} \rho\) \(\nRightarrow\) \(\xi|_U \approx_{\text{fl}} \rho|_U \wedge \xi|_V \approx_{\text{fl}} \rho|_V\)

**Proof.** The proofs of the two propositions are as follows:

- Clearly, every silent action of \(\xi\) is also a silent action of both \(\xi|_U\) and \(\xi|_U\), but the converse is, in general, false. Thus, the implication of the first proposition is easily seen. A counterexample for the reverse implication is given below:

  \[
  \begin{align*}
  \rho := \{ & x_1 & 0 \Box & 2 \Box & 4 \Box & 6 \Box & \ldots \\
  & x_2 & \Box & 1 \Box & 3 \Box & 5 \Box & 7 \Box & \ldots 
  \end{align*}
  \begin{align*}
  \xi := \{ & x_1 & 0 \Box & 2 \Box & 4 \Box & 6 \Box & \ldots \\
  & x_2 & \Box & 3 \Box & 5 \Box & 7 \Box & \ldots 
  \end{align*}
  \]
Defining $U := \{ x_1 \}$ and $V := \{ x_2 \}$, one can easily see that $\xi_{\{\}} \approx_{cl} \rho_{\{\}}$ and $\xi_{\{x_1\}} \approx_{cl} \rho_{\{x_1\}}$ hold, but $\xi \approx_{cl} \rho$ does not hold (e.g., since $\text{stfree}(\rho) \neq \text{stfree}(\xi)$).

- The proof of the second fact is obvious: Recall that $\Rightarrow$ are the same when all values $\square$ are ignored.

The above lemma reveals an important difference between the definitions $\approx_{cl}$ and $\approx_{cl}$ flows

---

\begin{itemize}
  \item $\text{stfree}(\chi) := \{\text{stfree}(\xi) \mid \xi \in \chi\}$
  \item $\text{flows}(\chi) := \{\text{flows}(\xi) \mid \xi \in \chi\}$
\end{itemize}

Finally, for a set of variable assignments $\chi$, we define:

\begin{itemize}
  \item $\text{CL}_{cl}(\chi) := \{\xi \mid \exists \rho \in \chi. \rho \approx_{cl} \xi\}$
  \item $\text{CL}_{cl}(\chi) := \{\xi \mid \exists \rho \in \chi. \rho \approx_{cl} \xi\}$
\end{itemize}

$\text{flows}(\xi)$ is sometimes called the desynchronization of $\xi$ since it contains the streams of values that appear when no values $\square$ are used. $\text{stfree}(\xi)$ is the stutter-free variant of $\xi$ where all silent actions are removed.

Note that we clearly have $\text{stfree}(\xi) \prec_{cl} \xi$ and $\text{flows}(\xi) \prec_{cl} \xi$, and moreover each of $\rho \prec_{cl} \text{stfree}(\xi)$ and $\rho \prec_{cl} \text{flows}(\xi)$ implies that $\rho = \xi$ holds since $\text{stfree}(\xi)$ and $\text{flows}(\xi)$ are minimal variable assignments with respect to $\prec_{cl}$ and $\prec_{cl}$, respectively. Because of this, we also see that the partial orders $\approx_{cl}$ and $\approx_{cl}$ are well-founded. We also clearly have $\rho \approx_{cl} \xi \iff \text{stfree}(\rho) = \text{stfree}(\xi)$ and $\rho \approx_{cl} \xi \iff \text{flows}(\rho) = \text{flows}(\xi)$ which gives an alternative definition of these equivalences.

By reflexivity of $\prec_{cl}$ and $\prec_{cl}$, we moreover conclude that $\xi \approx_{cl} \text{stfree}(\xi)$ and $\xi \approx_{cl} \text{flows}(\xi)$ holds, so that $\text{stfree}(\xi)$ and $\text{flows}(\xi)$ can be used as uniquely determined representatives of the equivalence classes $\text{CL}_{cl}(\{\xi\})$ and $\text{CL}_{cl}(\{\xi\})$, respectively. For this reason, we therefore have $\text{CL}_{cl}(\chi) = \text{CL}_{cl}(\text{stfree}(\chi))$ and $\text{CL}_{cl}(\chi) = \text{CL}_{cl}(\text{flows}(\chi))$, for example.

In many cases, it is assumed that a system $P$ is stuttering-closed, i.e., that $\text{Bhv}(P) = \text{CL}_{cl}(\text{Bhv}(P))$ holds. For example, it is proved in [157] that every
process of the language SIGNAL is stuttering-closed. We therefore easily see
the following refinement relations:

In the following, we will often not distinguish between stretch equivalent
assignments since these represent essentially the same computations that dif-
fer only by ‘silent actions’. There is a closely related definition of stuttering
states of transition systems that exactly reflects the intension of silent actions,
i.e., no action is performed by the system and no input values are consumed.
Memorized variables maintain their values and event variables have value $\Box$.
We therefore do not distinguish between stuttering equivalent computations
of a DPN, where a stuttering step is defined as one where no variable has a
value (different to $\Box$).

To reason about synchronous systems that are able to work in asyn-
chronous environments, we moreover define flow equivalences that are re-
stricted to input and output variables, respectively:

**Definition 7.45 (Input/Output Flow Equivalence).** For a DPN $P$ over the
variables $V = V_{in} \cup V_{loc} \cup V_{out}$, we introduce equivalence relations on
its variable assignments over $V$ by restricting flow equivalence to subsets of $U \subseteq V$:

- $\rho \approx_{in}^{fl} \xi : \iff \rho|_{U} \approx_{in} \xi|_{U}$
- $\rho \approx_{cl}^{fl} \xi : \iff \rho|_{U} \approx_{cl} \xi|_{U}$

and in particular

- $\rho \approx_{in}^{in} \xi : \iff \rho \approx_{in} V_{in} \xi$
- $\rho \approx_{out}^{out} \xi : \iff \rho \approx_{out} V_{out} \xi$

The equivalence relations $\approx_{in}^{in}$ and $\approx_{cl}$ allow us now to define endochronous
systems:

**Definition 7.46 (Endochronous Systems).** A synchronous DPN $P$ is called en-
dochronous if for all $\rho_1, \rho_2 \in \text{Bhv}(P)$ with $\rho_1 \approx_{in}^{in} \rho_2$, we also have $\rho_1 \approx_{cl} \rho_2$.

Recall that the motivation we obtained by our introductory examples was to
check whether we can uniquely resynchronize the input streams for a syn-
chronous system so that this synchronous system can be embedded in an
asynchronous environment with the help of another module that resynchro-
nizes the inputs and generates a clock tick for it. The above definition of
endochronous systems exactly have this property:

**Lemma 7.47 (Unique Resynchronization).** A synchronous module $P$ is en-
dochronous iff all synchronous behaviors of $P$ that are input flow equivalent to
a variable assignment $\xi$ over the variables of $P$ are stretch equivalent to each
other, i.e., $\forall \xi, \forall \rho_1, \rho_2 \in \text{Bhv}(P). \ \xi \approx_{in} \rho_1 \land \xi \approx_{in} \rho_2 \Rightarrow \rho_1 \approx_{in} \rho_2$. Thus, for an
endochronous module $P$, there is a resynchronization function $R_{P}$ that maps
every variable assignment $\xi$ that is flow equivalent to a behavior $\rho \in \text{Bhv}(P)$
to $\text{stfree}(\rho)$. We therefore have for every endochronous module $P$ the following
equation:

$$\text{CL}_{cl}(R_{P}(\text{flows}(\text{Bhv}(P)))) = \text{CL}_{cl}(\text{Bhv}(P))$$
Proof. Assume first that \( P \) is endochronous, and consider arbitrary variable assignments \( \xi \) and arbitrary \( \rho_1, \rho_2 \in \text{Bhv}(P) \) with \( \rho_1 \approx^m \xi \) and \( \rho_2 \approx^m \xi \). By transitivity of flow equivalence, it follows that \( \rho_1 \approx^m \rho_2 \) holds and by definition of endochrony that also \( \rho_1 \approx_{cl} \rho_2 \) holds. Now, assume that for any variable assignment \( \xi \) and all synchronous behaviors \( \rho_1, \rho_2 \in \text{Bhv}(P) \) with \( \rho_1 \approx^m \xi \) and \( \rho_2 \approx^m \xi \), we have \( \rho_1 \approx_{cl} \rho_2 \). For \( \xi = \rho_1 \), it then follows that \( P \) is endochronous.

Thus, each variable assignment \( \xi \) that is flow equivalent to a behavior \( \rho \in \text{Bhv}(P) \) can be associated with the class of stretch equivalent behaviors of \( P \) which defines the function \( \mathfrak{R}_P \) if \( \text{stfree}(\rho) \) is selected as representative of the class. \( \square \)

Thus, for endochronous systems, we can determine the synchronous input stream that has to be processed synchronously by the system, i.e., we can construct a module to resynchronize asynchronous input streams to the uniquely defined synchronous ones that can then be processed by the synchronous system. Note that the synchronous input streams are not really uniquely determined, since we only demand uniqueness up to stretch equivalence. This allows us, for example, to resynchronize the asynchronous input streams for the sequential-OR node specified in Figure 7.12.

Another lemma to characterize endochronous systems is the following one:

**Lemma 7.48.** If \( P \) and \( Q \) are endochronous systems over the same variables, then \( \text{CL}_{fl}(\text{Bhv}(P)) = \text{CL}_{fl}(\text{Bhv}(Q)) \) implies \( \text{CL}_{cl}(\text{Bhv}(P)) = \text{CL}_{cl}(\text{Bhv}(Q)) \).

Proof. Assume \( \rho \in \text{CL}_{cl}(\text{Bhv}(P)) \), so that \( \rho \in \text{CL}_{fl}(\text{Bhv}(P)) \), and by assumption also \( \rho \in \text{CL}_{fl}(\text{Bhv}(Q)) \) follows. Since \( Q \) is endochronous, there is a unique \( \rho' \in \text{Bhv}(Q) \) with \( \rho \approx_{fl} \rho' \). Hence, we also have \( \rho \in \text{CL}_{cl}(\text{Bhv}(Q)) \), so that \( \text{CL}_{cl}(\text{Bhv}(P)) \subseteq \text{CL}_{cl}(\text{Bhv}(Q)) \) follows. The reverse set inclusion follows analogously. \( \square \)

In order to understand the definition of endochrony, consider some examples. We have already seen that parallel-OR as shown in Figure 7.12 is not endochronous: The two behaviors shown in the figure are input flow equivalent, but not stretch equivalent. The sequential-OR function is however endochronous: To resynchronize input streams, simple remove all values \( \Box \) and use the remaining values in the obtained order for the reactions (remember that \( \Box \) is only used in silent actions of sequential-OR).

The process cpy2 shown in Figure 7.14 is not endochronous, since there are two behaviors \( \rho_1, \rho_2 \in \text{Bhv}(\text{cpy2}) \) with \( \rho_1 \approx^m \rho_2 \), but we do not have \( \rho_1 \approx_{cl} \rho_2 \). Clearly, this does not matter if we are only interested in flow equivalent output streams, but endochronous systems demand that the reactions to compute the output streams must be uniquely determined, which is not the case for cpy2.

This example motivates the definition of latency-insensitive systems, which is different to endochrony:
7.3 Globally Asynchronous Locally Synchronous Systems

Fig. 7.14. Module cpy2 is not endochronous.

Definition 7.49 (Latency-Insensitive Systems). A DPN $P$ is latency-insensitive if for all $\rho_1, \rho_2 \in \text{Bhv}(P)$ with $\rho_1 \approx_{\text{in}} \rho_2$, we also have $\rho_1 \approx_{\text{out}} \rho_2$.

Module cpy2 shown in Figure 7.14 is latency-insensitive, but not endochronous: We have $\xi \in \text{Bhv}(\text{cpy2})$ iff $\xi(x_1) = \xi(y_1)$ and $\xi(x_2) = \xi(y_2)$ holds. Thus, if two input flow equivalent behaviors $\rho_1, \rho_2 \in \text{Bhv}(P)$ must also be output flow equivalent. Hence, cpy2 is latency insensitive.

One can also consider a module cpy1 that copies just one input to a single output. In contrast to cpy2, cpy1 is endochronous since two input flow equivalent behaviors $\rho_1, \rho_2 \in \text{Bhv}(P)$ carry the same values different to □ in their input streams $\rho_i(x_1)$, and since $\rho_i(x_1) = \rho_i(y_1)$ holds for every behavior, we also have $\rho_i \approx_{\text{cl}} \rho_2$. Thus, cpy1 is both endochronous and latency-insensitive.

The modules cpy1 and cpy2 moreover show that the endochrony is not compositional: the composition $P \parallel Q$ of endochronous systems $P$ and $Q$ may not be endochronous (note that cpy2 = cpy1 \parallel cpy1). We will consider another example below where the composition is such that outputs of $P$ are fed as inputs to $Q$.

Hence, due to cpy2, there are latency-insensitive systems that are not endochronous. There is no example of an endochronous system that is not latency-insensitive, since we have the following theorem:

Theorem 7.50 (Endochrony implies Latency-Insensitivity).
Every endochronous system is latency-insensitive. There are however systems that are latency-insensitive, but not endochronous.

Proof. Assume that $P$ is an endochronous system, and that $\rho_1, \rho_2 \in \text{Bhv}(P)$ with $\rho_1 \approx_{\text{cl}} \rho_2$. Since $P$ is endochronous, $\rho_1 \approx_{\text{cl}} \rho_2$ follows, which implies $\rho_1 \approx_{\text{out}} \rho_2$, since for every output variable $y_i$ the streams $\rho_1(y_i)$ and $\rho_2(y_i)$ are obtained by stretching from the same □-free stream, thus they are also flow equivalent. Module cpy2 of Figure 7.14 is latency-insensitive, but not endochronous.
Since endochronous systems are special latency-insensitive systems, they map flow equivalent inputs to flow equivalent outputs, which is a mandatory requirement to be used in an asynchronous setting.

Clocks of streams are not latency-insensitive, and thus, not endochronous. Figure 7.15 shows the definition of the clock of a stream, i.e. a boolean stream that is 1 whenever the input stream is different to $\Box$. As can be seen, the listed behaviors $\rho_1, \rho_2 \in \text{Bhv}(\text{clock})$ carry the same values different to $\Box$ in their input streams $\rho_1(x)$, hence, we have $\rho_1 \approx_{\text{fl}}^\text{in} \rho_2$. However, we do not have $\rho_1 \approx_{\text{fl}}^\text{out} \rho_2$, and therefore clocks are not latency-insensitive, and therefore also not endochronous.

Note that if we would allow endochronous systems to inspect also their outputs, i.e., if the definition of endochrony would be such that a system $P$ would be endochronous if for all $\rho_1, \rho_2 \in \text{Bhv}(P)$ with $\rho_1 \approx_{\text{fl}}^\text{in} \rho_2$, we also have $\rho_1 \approx_{\text{fl}}^\text{out} \rho_2$, then clocks would become endochronous: To resynchronize input streams, we could inspect the output stream $y$ and would know due to the occurrences of 0 where we have to insert values $\Box$. However, as we have to resynchronize the inputs before the synchronous system can perform an action to produce the outputs, this definition would not be useful in practice.

We have already discussed sequential functions, which are functions where the firing rules are determined by sequentially reading one input value required for the reaction one after the other. For a sequential function, we therefore always start with an initial set of input variables $V_0$ that are read first, and depending on the values read from these variables, another set of variables $V_1$ is read next until all input values required for the reaction are known. It is not difficult to see that the following fact holds:

**Theorem 7.51 (Sequentiality implies Endochrony).**

Every sequential system is endochronous, but there are endochronous systems that are not sequential.

**Proof.** We can resynchronize a desynchronized input stream by following the sequential algorithm to read input values of the first reaction. We then insert $\Box$ for all input variables where no values are required for the first reaction. After generating values for the output values (and assigning $\Box$ to those output variables where no other value is written to), we can proceed in the same manner to schedule the input values used for the next reaction, and so on.
The variant of Berry’s Gustave function shown in Figure 7.16 is endochronous, but not sequential: The function is endochronous since every desynchronized input stream can be resynchronized into reactions as follows: Assume that all values □ are removed from the input streams. Then, we consider the first three values of the input streams $x_1$, $x_2$, and $x_3$. For all possible eight values, there is a unique action that can be chosen, and this reaction might require all three or only two of the three values. In case only two values are required, we insert □ for the third, and proceed with the remaining values in the same manner. Thus, function Gustave is endochronous. It is however not sequential, since the asynchronous version where □ values are not used does not allow us to definitely start with an input to be read (for each input, there are reactions where that input is not required).

The original definition of endochrony given in [27, 30, 32, 33] demands that for endochronous systems, there is a way to derive from the knowledge that a certain set of input variables are present and of their actual values that another set of input variables must also be present. Starting from the empty set of known variables, it must then be possible for an endochronous system to derive the presence and if present, also the values, of all variables. It is then proved that systems that have this property and only these systems have the ability to reconstruct the corresponding synchronous run if these became desynchronized. The example shown in Figure 7.16 demonstrates however that this is not true. Hence, this example seems to be counterexample to the theorem given in [27, 30, 32, 33], but this may also be just the result of a misunderstanding of the author, since many definitions given in [27, 30, 32, 33] (as the derivation of values of other values) are kept informal, and can therefore not really be checked.

We should also emphasize that sequential systems have input streams that have a master clock, i.e., input streams that have present values whenever one stream of the system has a value. This is due to the fact that a sequential function has to start each reaction by reading an initial set of input variables (that therefore must be present in each reaction). Inputs with the master clock therefore provide implicitly a clock signal for the sequential system.
merge

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 :: $A$)</td>
<td>(b :: $B$)</td>
<td>(□ :: $C$)</td>
<td>b \neq □ [b]</td>
</tr>
<tr>
<td>(0 :: $A$)</td>
<td>(□ :: $B$)</td>
<td>(c :: $C$)</td>
<td>c \neq □ [c]</td>
</tr>
<tr>
<td>(□ :: $A$)</td>
<td>(□ :: $B$)</td>
<td>(□ :: $C$)</td>
<td>□</td>
</tr>
</tbody>
</table>

\[
\begin{aligned}
z_1 &= x_2$

z_2 &= 1 \text{ when } (x_2 \neq z_1 \land x_2 \neq □ \land z_1 \neq □) \\
y_1 &= \text{merge}(x_1, z_2, x_3)
\end{aligned}
\]

Fig. 7.17. A DPN whose components are endochronous, but their composition is not.

Fig. 7.18. Embedding an Endochronous System in an Asynchronous Environment using Wrappers

To summarize, endochrony is an important property for a synchronous system that allows one to embed the synchronous system in an asynchronous environment. Since endochrony means that for every endochronous component $P$, there is a resynchronization function $R_P$, the idea is simply to implement $R_P$ in a wrapper module in front of $P$ that collects all inputs until the inputs of the next reaction of $P$ are available and then sends $P$ a clock signal so that $P$ executes the next reaction. Since values □ are not used in asynchronous environments (recall their meaning is just to keep streams assigned to different variables aligned in synchronous reactions), $R_P$ will also produce values □ if necessary. Analogously, we can omit the production of these values by $P$. As a result, we can implement for every endochronous component $P$ an ‘asynchronous system’ composed of $R_P$ and $P$ that implements the behaviors flows($\text{Bhv}(P)$).

Even though the problem to embed synchronous components in asynchronous environments is solved in theory by endochrony, there are further practical problems: The most important one is that the synchronous com-
ponents should be as small as possible to increase the efficiency due to asynchronous computations as far as possible. For example, consider a synchronous system \( P \parallel Q \) composed of two synchronous components \( P \) and \( Q \). If \( P \parallel Q \) would be endochronous, we could of course proceed as mentioned above by adding a wrapper for the entire synchronous system \( P \parallel Q \) to implement \( \text{flows(Bhv}(P \parallel Q)) \). It would however be more efficient to embed the components \( P \) and \( Q \) separately and to implement their composition/communication in an asynchronous manner. Of course, it is required that both \( P \) and \( Q \) are endochronous so that it makes no real difference that we implement \( \text{flows(Bhv}(P)) \) and \( \text{flows(Bhv}(Q)) \) instead of \( P \) and \( Q \), respectively.

However, it is unfortunately not the case that endochrony of \( P \) and \( Q \) ensure that also \( P \parallel Q \) is endochronous, as we will show below by an example, thus endochrony is not closed under parallel composition. We have already seen this by module cpy2 which can be obtained as parallel composition of two modules cpy1, but the following example taken from [245, 246] is more interesting and leads to further definitions. The example DPN is shown in Figure 7.17 and makes use of the input variables \( \mathcal{V}_\text{in} = \{x\} \), local variables \( \mathcal{V}_\text{loc} = \{z_1, z_2\} \), and output variables \( \mathcal{V}_\text{out} = \{y_1\} \). Local variable \( z_1 \) simply delays the input stream \( x_2 \) by one reaction, but has the same clock as \( x_2 \). Thus, \( z_2 \) is 1 whenever \( x_2 \) changed its value (where all values \( \Box \) are ignored).

Finally, \( y_1 \) is obtained according to the firing rules of the merge node, i.e.,

\[
y_1 = \begin{cases} 
  z_2 : & \text{if } x_1 = 1 \\
  x_3 : & \text{if } x_1 = 0 \\
  \Box : & \text{if } x_1 = \Box 
\end{cases}
\]

Process merge is easily seen to be sequential, and therefore it is endochronous. Also the other two nodes are endochronous. We might consider the first two equations that define the local variables \( z_1 \) and \( z_2 \) as a single process filter that emits 1 whenever a change of values different to \( \Box \) is seen on input stream \( x_2 \). Also process filter is endochronous.

However, the composition of merge and filter is not endochronous, since we have the two input flow equivalent behaviors \( \rho_1 \) and \( \rho_2 \) that are not stretch equivalent:

| \( \rho_1(x_1) \) | 0 1 \( \Box \) 1 0 ... |
| \( \rho_1(x_2) \) | 1 0 0 1 \( \Box \) ... |
| \( \rho_1(x_3) \) | 1 \( \Box \) \( \Box \) \( \Box \) 0 ... |
| \( \rho_1(z_1) \) | \( \Box \) 1 0 0 \( \Box \) ... |
| \( \rho_1(z_2) \) | \( \Box \) 1 \( \Box \) \( \Box \) ... |
| \( \rho_1(y_1) \) | 1 1 \( \Box \) 1 0 ... |
| \( \rho_2(x_1) \) | \( \Box \) 0 1 \( \Box \) 1 0 ... |
| \( \rho_2(x_2) \) | 1 \( \Box \) 0 0 1 \( \Box \) ... |
| \( \rho_2(x_3) \) | \( \Box \) 1 \( \Box \) \( \Box \) \( \Box \) 0 ... |
| \( \rho_2(z_1) \) | \( \Box \) \( \Box \) 1 0 0 \( \Box \) ... |
| \( \rho_2(z_2) \) | \( \Box \) \( \Box \) 1 \( \Box \) \( \Box \) ... |
| \( \rho_2(y_1) \) | 1 1 \( \Box \) 1 0 ... |
It can be easily seen that both variable assignments $\rho_1, \rho_2$ are behaviors of the synchronous DPN. It is moreover easily seen that $\rho_1 \approx_{fl} \rho_2$ holds, but that $\rho_1 \approx_{cl} \rho_2$ does not hold. Each component fires with its own clock that is determined by its input flows since each component is endochronous and can therefore determine its clock on its own by its inputs. As can be seen, the first reaction of $\rho_1$ (where all components fire) is split into single reactions in $\rho_2$ where some components fire while the others are idle (stuttering). Note that merge does not require the output of filter in its first reaction so that it can fire before, after or at the same time as the first reaction of filter.

Hence, the composition $P \parallel Q$ of two endochronous components $P$ and $Q$ may not be endochronous, which prevents us at the moment from embedding such an entire system in an asynchronous environment. However, analyzing the problem in more detail will give us another criterion to ensure the correctness in these cases: Recall that endochrony allows us to generate an asynchronous process $\text{flows}(\text{Bhv}(P))$ from a synchronous implementation $P$ by means of the resynchronization function $R_P$ that can be implemented in a separate wrapper or can be merged in a common code generation. If $P \parallel Q$ would also be endochronous, we could proceed in the same way with the entire system, i.e., adding a wrapper for $P \parallel Q$ to implement the asynchronous system $\text{flows}(\text{Bhv}(P \parallel Q))$. If $P \parallel Q$ is not endochronous, this might not be correct. However, if the desynchronization function would commute with parallel composition, i.e., if $\text{flows}(\text{Bhv}(P \parallel Q)) = \text{flows}(\text{Bhv}(P)) \parallel_{fl} \text{flows}(\text{Bhv}(Q))$ would hold for asynchronous parallel composition $\parallel_{fl}$ defined below, then endochrony of $P$ and $Q$ would solve the remaining problem (since we can implement $\text{flows}(\text{Bhv}(P))$ and $\text{flows}(\text{Bhv}(Q))$ in this case). For this reason, isochronous composition of systems are defined below. To this end, we start with the definition of synchronous and asynchronous composition of processes:

**Definition 7.52 (Synchronous/Asynchronous Composition).** Given two sets of variable assignments $P$ and $Q$ over variables $U$ and $V$, respectively, we define their synchronous and asynchronous compositions as follows:

- $P \parallel Q := \{ \xi \mid \xi_U \in P \land \xi_V \in Q \}$
- $P \parallel_{fl} Q := \{ \xi \mid \xi_U \in \text{flows}(P) \land \xi_V \in \text{flows}(Q) \}$

Given modules $P$ and $Q$ over variables $U$ and $V$, we clearly have $\text{Bhv}(P \parallel Q) := \text{Bhv}(P) \parallel \text{Bhv}(Q)$ for the parallel composition $P \parallel Q$. We moreover define a new process $P \parallel_{fl} Q$ whose semantics is defined as follows:

- $\text{Bhv}(P \parallel_{fl} Q) := \text{Bhv}(P) \parallel_{fl} \text{Bhv}(Q)$

It is easily seen that $P \parallel_{fl} Q = \text{flows}(P) \parallel_{fl} \text{flows}(Q)$ holds. Thus, each behavior $\xi \in (P \parallel Q)$ of the synchronous composition $P \parallel Q$ consists of a behavior $\xi_U \in P$ of $P$ and $\xi_V \in Q$ of $Q$. Note that these behaviors make use of $\Box$, so that the streams assigned by $\xi$ to the variables in $U \cup V$ are synchronized. In contrast, behaviors $\xi \in (P \parallel_{fl} Q)$ of the asynchronous composition $P \parallel_{fl} Q$ ignore occurrences of $\Box$, so that the communication is done asynchronously. As
a consequence, \( P \parallel Q \) considers all values including \( \Box \) for composing behaviors of \( P \) and \( Q \), while \( P \parallel_{\Box} Q \) only considers values different to \( \Box \) for this purpose. For isochronous systems, this makes no difference:

**Definition 7.53 (Isochronous Composition of Systems).**

The parallel composition of synchronous modules \( P \) and \( Q \) is isochronous if the following holds: \( \text{flows}(\text{Bhv}(P \parallel Q)) = \text{flows}(\text{Bhv}(P)) \parallel_{\Box} \text{flows}(\text{Bhv}(Q)) \). Analogously, the parallel composition of sets of variable assignments \( P \) and \( Q \) is isochronous if \( \text{flows}(P \parallel Q) = \text{flows}(P) \parallel_{\Box} \text{flows}(Q) \) holds.

As can be seen, isochrony just means that desynchronization performed by \( \text{flows}(\cdot) \) commutes with parallel composition. Due to synchronous and asynchronous environments, we use synchronous parallel composition \( \parallel \) before desynchronization and asynchronous parallel composition \( \parallel_{\Box} \) after desynchronization. However, as we will see in the next lemma, the use of asynchronous parallel composition \( \parallel_{\Box} \) is not really required since it is applied to already desynchronized behaviors (note that the two parallel compositions only differ in that \( \parallel_{\Box} \) first desynchronizes the behaviors, so that occurrences of \( \Box \) are ignored):

**Lemma 7.54 (Isochronous Systems).**

For all sets of variable assignments \( P \) and \( Q \), the following are equivalent:

1. parallel composition of \( P \) and \( Q \) is isochronous
2. \( \text{flows}(P) \parallel_{\Box} \text{flows}(Q) = \text{flows}(P \parallel Q) \)
3. \( \text{flows}(P) \parallel \text{flows}(Q) = \text{flows}(P \parallel Q) \)
4. \( (P \parallel_{\Box} Q) = \text{flows}(P \parallel Q) \)
5. \( \text{flows}(P) \parallel_{\Box} \text{flows}(Q) \subseteq \text{flows}(P \parallel Q) \)
6. \( \text{flows}(P) \parallel \text{flows}(Q) \subseteq \text{flows}(P \parallel Q) \)
7. \( (P \parallel_{\Box} Q) \subseteq \text{flows}(P \parallel Q) \)
8. \( \forall \rho \in P. \forall \eta \in Q. \rho \approx_{U \cap V}^{U \cap V} \eta \rightarrow \exists \zeta \in (P \parallel Q). \zeta \approx_{U}^{U \cap V} \rho \land \zeta \approx_{V}^{U \cap V} \eta \)
9. \( \forall \rho \in P. \forall \eta \in Q. \rho \approx_{U \cap V}^{U \cap V} \eta \rightarrow \text{join}(\rho, \eta) \in \text{flows}(P \parallel Q) \),

where \( \text{join}(\rho, \eta)(x) := \begin{cases} \text{flows}(\rho)(x) & : x \in U \\ \text{flows}(\eta)(x) & : x \in V \end{cases} \)

**Proof.** The proof of the equivalences is given by the following equivalence proofs:

- The equivalence of 1 and 2 is the definition of isochrony.
- We prove \( \text{flows}(P) \parallel_{\Box} \text{flows}(Q) = \text{flows}(P) \parallel \text{flows}(Q) \) as follows (where we use the fact that \( \text{flows}(\text{flows}(M)) = \text{flows}(M) \) holds for every \( M \)):

\[
\xi \in \text{flows}(P) \parallel_{\Box} \text{flows}(Q) \\
\iff \xi_{|U} \in \text{flows}(\text{flows}(P)) \land \xi_{|V} \in \text{flows}(\text{flows}(Q)) \\
\iff \xi_{|U} \in \text{flows}(P) \land \xi_{|V} \in \text{flows}(Q) \\
\iff \xi \in \text{flows}(P) \parallel \text{flows}(Q)
\]

Thus, 2 and 3 are equivalent to each other, as well as 5 and 6.
We prove \( \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \subseteq \text{flows}(\mathcal{P}) \parallel \text{flows}(\mathcal{Q}) \), so that 3 is equivalent to 6. The proof is obtained as follows, where the steps (1) and (2) need additional explanations: (1) holds since \( \xi \) and \( \zeta \) are variable assignments over the variables \( \mathcal{U} \cup \mathcal{V} \), and therefore \( \xi = \text{flows}(\zeta) \) is equivalent to \( \xi_{\mathcal{U}} = \text{flows}(\zeta_{\mathcal{U}}) \land \xi_{\mathcal{V}} = \text{flows}(\zeta_{\mathcal{V}}) \) (note Lemma 7.43). Moreover, (2) follows by using the witnesses \( \rho := \xi_{\mathcal{U}} \) and \( \eta := \xi_{\mathcal{V}} \):

\[
\xi \in \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \quad \iff \quad \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \xi = \text{flows}(\zeta) \\
\iff \quad \exists \xi. \xi_{\mathcal{U}} \in \mathcal{P} \land \xi_{\mathcal{V}} \in \mathcal{Q} \land \xi = \text{flows}(\zeta) \\
(1) \quad \exists \xi. \xi_{\mathcal{U}} \in \mathcal{P} \land \xi_{\mathcal{V}} \in \mathcal{Q} \land \xi_{\mathcal{U}} = \text{flows}(\zeta_{\mathcal{U}}) \land \xi_{\mathcal{V}} = \text{flows}(\zeta_{\mathcal{V}}) \\
(2) \quad \exists \rho \in \mathcal{P}. \exists \eta \in \mathcal{Q}. \rho \in \mathcal{P} \land \eta \in \mathcal{Q} \land \xi_{\mathcal{U}} = \text{flows}(\rho) \land \xi_{\mathcal{V}} = \text{flows}(\eta) \\
\Rightarrow \quad \exists \rho \in \mathcal{P}. \exists \eta \in \mathcal{Q}. \xi_{\mathcal{U}} = \text{flows}(\rho) \land \xi_{\mathcal{V}} = \text{flows}(\eta) \\
\Rightarrow \quad \xi_{\mathcal{U}} \in \text{flows}(\mathcal{P}) \land \xi_{\mathcal{V}} \in \text{flows}(\mathcal{Q}) \\
\Rightarrow \quad \xi \in \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \\
\]

Hence, properties 1-7 are equivalent to each other. We proceed by proving the equivalence of properties 6 and 8:

\[
\text{flows}(\mathcal{P}) \parallel \text{flows}(\mathcal{Q}) \subseteq \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \\
\iff \quad \forall \xi. \xi \in (\text{flows}(\mathcal{P}) \parallel \text{flows}(\mathcal{Q})) \rightarrow \xi \in \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \\
\iff \quad \forall \xi. \xi_{\mathcal{U}} \in \text{flows}(\mathcal{P}) \land \xi_{\mathcal{V}} \in \text{flows}(\mathcal{Q}) \rightarrow \xi \in \text{flows}(\mathcal{P} \parallel \mathcal{Q}) \\
\iff \quad \forall \xi. (\exists \rho \in \mathcal{P}. \xi_{\mathcal{U}} = \text{flows}(\rho)) \land (\exists \eta \in \mathcal{Q}. \xi_{\mathcal{V}} = \text{flows}(\eta)) \\
\quad \rightarrow \quad \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \xi = \text{flows}(\zeta) \\
\iff \quad \forall \rho \in \mathcal{P}. \forall \eta \in \mathcal{Q}. \forall \xi. \\
\quad \xi_{\mathcal{U}} = \text{flows}(\rho) \land \xi_{\mathcal{V}} = \text{flows}(\eta) \rightarrow \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \xi = \text{flows}(\zeta) \\
\iff \quad \forall \rho \in \mathcal{P}. \forall \eta \in \mathcal{Q}. \forall \xi. \\
\quad \xi_{\mathcal{U}} = \text{flows}(\rho) \land \xi_{\mathcal{V}} = \text{flows}(\eta) \rightarrow \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \text{flows}(\rho) = \text{flows}(\xi_{\mathcal{U}}) \land \text{flows}(\eta) = \text{flows}(\xi_{\mathcal{V}}) \\
\iff \quad \forall \rho \in \mathcal{P}. \forall \eta \in \mathcal{Q}. \\
\quad (\exists \xi. \xi_{\mathcal{U}} = \text{flows}(\rho) \land \xi_{\mathcal{V}} = \text{flows}(\eta)) \\
\quad \rightarrow \quad \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \text{flows}(\rho) = \text{flows}(\xi_{\mathcal{U}}) \land \text{flows}(\eta) = \text{flows}(\xi_{\mathcal{V}}) \\
\iff \quad \forall \rho \in \mathcal{P}. \forall \eta \in \mathcal{Q}. \\
\quad \rho \approx_{\mathcal{U} \cap \mathcal{V}} \zeta \\
\quad \rightarrow \quad \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \text{flows}(\rho) = \text{flows}(\xi_{\mathcal{U}}) \land \text{flows}(\eta) = \text{flows}(\xi_{\mathcal{V}}) \\
\iff \quad \forall \rho \in \mathcal{P}. \forall \eta \in \mathcal{Q}. \\
\quad \rho \approx_{\mathcal{U} \cap \mathcal{V}} \eta \rightarrow \exists \xi \in (\mathcal{P} \parallel \mathcal{Q}). \zeta \approx_{\mathcal{U}} \rho \land \zeta \approx_{\mathcal{V}} \eta \\
\]

Finally, we prove the equivalence between properties 8 and 9: To this end, we consider arbitrary \( \rho \in \mathcal{P} \) and \( \eta \in \mathcal{Q} \) with \( \rho \approx_{\mathcal{U} \cap \mathcal{V}} \eta \). Then, we have
the following (note that by definition of join(ρ, η), we have join(ρ, η)|U = flows(ρ) and join(ρ, η)|V = flows(η)):

\[
\begin{align*}
\text{join}(\rho, \eta) \in \text{flows}(P \parallel Q) & \iff \exists \zeta \in (P \parallel Q). \text{join}(\rho, \eta) = \text{flows}(\zeta) \\
& \iff \exists \zeta \in (P \parallel Q). \text{join}(\rho, \eta)|U = \text{flows}(\zeta|U) \land \text{join}(\rho, \eta)|V = \text{flows}(\zeta|V) \\
& \iff \exists \zeta \in (P \parallel Q). \text{flows}(\rho) = \text{flows}(\zeta|U) \land \text{flows}(\eta) = \text{flows}(\zeta|V) \\
& \iff \exists \zeta \in (P \parallel Q). \rho \approx^U \zeta \land \eta \approx^V \zeta
\end{align*}
\]

The last two properties are the most interesting ones of the above equivalent characterizations of isochrony, since the other ones just deal with certain kinds of redundancies of the definitions. Note that \( \rho \approx^U U \cap V \eta \) means that \( \rho \) and \( \eta \) can be asynchronously combined since they agree on the flows of the shared variables. Hence, an isochronous composition \( P \parallel Q \) requires for all behaviors \( \rho \in P \) and \( \eta \in Q \) that can be asynchronously combined that there is a behavior \( \zeta \) of the synchronous composition \( P \parallel Q \) that is flow equivalent to \( \rho \) and \( \zeta \) on the shared variables. Hence, isochrony makes sure that replacing a synchronous parallel composition by an asynchronous one does not generate new behaviors.

The last characterization of Lemma 7.54 can also be used to check whether a composition \( P \parallel Q \) is isochronous. To this end, one has to consider arbitrary behaviors \( \rho \in \text{Bhv}(P) \) and \( \eta \in \text{Bhv}(Q) \) that are flow equivalent on the shared variables. Then, a synchronous behavior \( \zeta \in \text{Bhv}(P \parallel Q) \) must be constructed from \( \rho \) and \( \eta \).

For example, consider the system shown in Figure 7.17 whose components filter and merge are endochronous, but their composition is not endochronous:

\[
\begin{align*}
\begin{cases}
z_2 &= \text{filter}(x_2) \\
y_1 &= \text{merge}(x_1, z_2, x_3)
\end{cases}
\end{align*}
\]

The system has inputs \( x_1, x_2, x_3 \), outputs \( y_1 \), and the local variable \( z_2 \). The only shared variable is the local variable \( z_2 \). Any input synchronous sequence for \( x_2 \) generates a synchronous sequence for \( z_2 \) consisting only of values 0 and 1 where 1 occurs at positions where \( x_2 \) has a value 1 or 0 that is different to the last boolean value. It is not difficult to see that we can construct a behavior \( \eta \) of merge that consumes these values whenever \( x_2 = 1 \) holds. Thus, the composition of filter and merge is isochronous, but the entire system is not endochronous. Due to the above theorem, we can therefore easily generate an asynchronous implementation of this synchronous system.

Since the definitions of endochrony and isochrony are given in terms of behaviors, checking endochrony and isochrony by computer programs is not straightforward. For endochrony, we have already proved that every sequential component is endochronous, and checking sequentiality is not difficult for dataflow programs. For isochrony, we can make use of the criterion given in the theorem below as outlined after the theorem [30]: Recall that the problem is to check that for all \( \rho \in P \) and \( \eta \in Q \) that can be 'asynchronously...
combined’, i.e., where \( \rho \approx_{(\mathcal{U} \cap \mathcal{V})}^{\mathcal{U} \cap \mathcal{V}} \eta \) holds, there is a corresponding synchronous behavior \( \zeta \in (\mathcal{P} \parallel \mathcal{Q}) \), i.e., where \( \zeta \approx_{\mathcal{U} \cap \mathcal{V}}^{\mathcal{U} \cap \mathcal{V}} \rho \) and \( \zeta \approx_{\mathcal{U} \cap \mathcal{V}}^{\mathcal{U} \cap \mathcal{V}} \eta \) holds. The problem is thereby that it may not be possible to synchronously combine \( \rho \in \mathcal{P} \) and \( \eta \in \mathcal{Q} \) (since \( \rho|_{\mathcal{V}} \neq \eta|_{\mathcal{U}} \)), but we might be successful in synchronously combining stretch equivalent behaviors \( \rho' \in \mathcal{P} \) and \( \eta' \in \mathcal{Q} \). The theorem below lists a condition that ensures this:

**Theorem 7.55 (Checking Isochrony).** Consider the following properties (A), (B) and (C) at a position \( t \in \mathbb{N} \) of synchronous behaviors \( \rho \) and \( \eta \) of systems \( \mathcal{P} \) and \( \mathcal{Q} \) on variables \( \mathcal{U} \) and \( \mathcal{V} \), respectively:

(A) Neither \( \rho \) nor \( \eta \) is silent on the shared variables \( \mathcal{U} \cap \mathcal{V} \), i.e.

\[
\exists x \in \mathcal{U} \cap \mathcal{V}. \quad \rho(x)^{(t)} \neq \square \\
\exists x \in \mathcal{U} \cap \mathcal{V}. \quad \eta(x)^{(t)} \neq \square
\]

(B) \( \rho \) and \( \eta \) agree on the present shared variables \( \mathcal{U} \cap \mathcal{V} \), i.e.

\[
\forall x \in \mathcal{U} \cap \mathcal{V}. \quad \rho(x)^{(t)} \neq \square \land \eta(x)^{(t)} \neq \square \rightarrow \rho(x)^{(t)} = \eta(x)^{(t)}
\]

(C) \( \rho \) and \( \eta \) agree on all shared variables \( \mathcal{U} \cap \mathcal{V} \), i.e.

\[
\forall x \in \mathcal{U} \cap \mathcal{V}. \quad \rho(x)^{(t)} = \eta(x)^{(t)}
\]

\( \mathcal{P} \parallel \mathcal{Q} \) is isochronous if for all \( \rho \in \mathcal{P} \) and \( \eta \in \mathcal{Q} \) with \( \rho \approx_{(\mathcal{U} \cap \mathcal{V})}^{\mathcal{U} \cap \mathcal{V}} \eta \) properties (A) and (B) imply (C) for every position \( t \in \mathbb{N} \). The converse is true if \( \mathcal{P} \) and \( \mathcal{Q} \) are additionally endochronous.

**Proof.** First assume that for all \( \rho \in \mathcal{P} \) and \( \eta \in \mathcal{Q} \), properties (A) and (B) imply (C), so that we have to prove that \( \mathcal{P} \parallel \mathcal{Q} \) is isochronous. According to Lemma 7.54, this means that we have to prove that there is a synchronous behavior \( \zeta \in (\mathcal{P} \parallel \mathcal{Q}) \) with \( \zeta \approx_{\mathcal{U} \cap \mathcal{V}}^{\mathcal{U} \cap \mathcal{V}} \rho \) and \( \zeta \approx_{\mathcal{U} \cap \mathcal{V}}^{\mathcal{U} \cap \mathcal{V}} \eta \) provided that \( \rho \approx_{(\mathcal{U} \cap \mathcal{V})}^{(\mathcal{U} \cap \mathcal{V})} \eta \) holds. We construct \( \zeta \) by following \( \rho \) and \( \eta \) step-by-step as follows, maintaining current positions \( t_\rho, t_\eta \) and \( t_\zeta \) where we distinguish between the following cases:

**Case 1:** Both behaviors \( \rho(t_\rho) \) and \( \eta(t_\eta) \) are silent on all shared variables. Since the reactions \( \rho(t_\rho) \) and \( \eta(t_\eta) \) therefore agree on all shared variables \( \mathcal{U} \cap \mathcal{V} \), they can be synchronously combined, and we define \( \zeta(t_\zeta) \) as follows:

\[
\zeta(x)^{(t_\zeta)} := \begin{cases} 
\rho(x)^{(t_\rho)} & : x \in \mathcal{U} \\
\eta(x)^{(t_\eta)} & : x \in \mathcal{V}
\end{cases}
\]

We then increase all positions: \( t_\zeta := t_\zeta + 1 \), \( t_\rho := t_\rho + 1 \), and \( t_\eta := t_\eta + 1 \) to determine the next reaction of \( \zeta \).

**Case 2:** Only \( \rho(t_\rho) \) is silent on all shared variables, while there is a variable \( x \in \mathcal{U} \cap \mathcal{V} \) with \( \eta(x)^{(t_\eta)} \neq \square \). For this reason, the reactions \( \rho(t_\rho) \) and \( \eta(t_\eta) \) cannot be synchronously combined. For this reason, we perform a stuttering step of \( \mathcal{Q} \) to defer the reaction of \( \eta \) into the future w.r.t. \( \rho \), and we define
\[ \zeta(x)^{(t_\zeta)} := \begin{cases} 
\rho(x)^{(t_\rho)} : & \text{if } x \in U \\
\square : & \text{if } x \in V \setminus U 
\end{cases} \]

We increase the time positions as follows: \( t_\zeta := t_\zeta + 1 \), \( t_\rho := t_\rho + 1 \), and \( t_\eta := t_\eta \).

Case 3: Only \( \eta^{(t_\eta)} \) is silent on all shared variables, while there is a variable \( x \in U \cap V \) with \( \rho(x)^{(t_\rho)} \neq \square \). This case is handled analogously as the previous case.

Case 4: None of the behaviors \( \rho^{(t_\rho)} \) and \( \eta^{(t_\eta)} \) is silent on all shared variables.

Note that they must agree on the present variables, since we otherwise would not have \( \rho \approx_{U \cap V} \eta \). Therefore, conditions (A) and (B) listed in the theorem are satisfied, and therefore, we also have condition (C). Hence, the reactions \( \rho^{(t_\rho)} \) and \( \eta^{(t_\eta)} \) agree on all shared variables, and can therefore be synchronously combined. We therefore define:

\[ \zeta(x)^{(t_\zeta)} := \begin{cases} 
\rho(x)^{(t_\rho)} : & \text{if } x \in U \\
\eta(x)^{(t_\eta)} : & \text{if } x \in V 
\end{cases} \]

and we increase all positions: \( t_\zeta := t_\zeta + 1 \), \( t_\rho := t_\rho + 1 \), and \( t_\eta := t_\eta + 1 \).

Repeating the above construction of \( \zeta(x)^{(t_\zeta)} \) determines a behavior \( \zeta \in P \parallel Q \) as required for isochrony.

Next, assume that \( P \) and \( Q \) are both endochronous, and we have behaviors \( \rho \in P \) and \( \eta \in Q \) with the property \( \rho \approx_{U \cap V} \eta \) where at some position \( t \in \mathbb{N} \) condition (A) and (B) hold, but (C) is false. Hence, there is a shared variable \( x \in U \cap V \) that is absent in one behavior, but present in the other one. For this reason, \( \rho \in P \) and \( \eta \in Q \) cannot be synchronously combined. Since \( P \) and \( Q \) are endochronous, \( \rho \) and \( \eta \) are (up to stretch equivalence) the only behaviors whose desynchronizations are \( \text{flows}(\rho) \) and \( \text{flows}(\eta) \), respectively. Therefore, the behavior \( \zeta \in (P \parallel Q) \) that must exist due to the isochrony of \( P \parallel Q \) must be obtained by synchronous composition of behaviors \( \rho' \in P \) and \( \eta' \in Q \) that are stretch equivalent to \( \rho \in P \) and \( \eta \in Q \), respectively. However, none of these behaviors \( \rho' \in P \) and \( \eta' \in Q \) can be synchronously combined, since at time the conditions (A) and (B) hold, but not (C). Inserting silent actions does not help here, it just defers the same situation to a future point of time. Thus, \( \zeta \) can not exist, and therefore \( P \parallel Q \) is not isochronous. \( \square \)

The above theorem is important for checking isochrony, since we can check whether (A) and (B) implies (C) for each pair of behaviors \( (\rho, \eta) \in P \times Q \) during a simultaneous depth-first traversals on the state transition diagrams of \( P \) and \( Q \). Whenever new states \( s_P \) and \( s_Q \) reached in the transition systems of \( P \) and \( Q \), respectively, the prefixes of \( \rho \in P \) and \( \eta \in Q \) are both extended by a new reaction. If the reactions \( s_P \) and \( s_Q \) do not agree on the present shared variables, we can successfully terminate, since all behaviors that could be obtained by further extending the so far seen prefixes were not flow equivalent. In all other cases, we check which of the cases listed in the above proof...
applies, successfully terminate in the cases 1-3 and check whether (C) holds in the remaining case 4.

Thus, the criterion given in the above theorem can be easily checked by means of model-checking. In the only interesting case, where the components are endochronous, this check is even equivalent to isochrony. At this stage, we should remark that the condition listed in the above theorem has been used as definition of isochrony in [27, 30, 32, 33], while our definition of isochrony (that is essentially taken from [157, 245, 246]) was not given a name in these references. The important property is however that the asynchronous composition does not generate more behaviors as the synchronous composition, so that our definition of isochrony seems to be more directed towards the application, while the above theorem proves both definitions equivalent for endochronous systems.

An important special case is moreover the following: Recall that the parallel composition $P ∥ Q$ of synchronous systems $P$ and $Q$ over disjoint sets of variables is not endochronous (since the two subsystems can fire independent of each other). For this reason, we would have no chance to embed such systems in asynchronous environments if we would only be able to exploit endochrony. Fortunately, all these systems are isochronous, so that the above theorem allows us to implement them asynchronously which is natural due to the lack of interaction between the subsystems $P$ and $Q$:

**Corollary 7.56 (Isochrony of Independent Systems).** The parallel composition $P ∥ Q$ of synchronous systems $P$ and $Q$ over disjoint sets of variables is isochronous.

**Proof.** If $U \cap V = \emptyset$ holds, then $\rho \approx_{U \cap V} \eta$ holds for every $\rho$ and $\eta$. Thus, isochrony reduces to $\forall \rho \in P. \forall \eta \in Q. \exists \zeta \in (P ∥ Q). \zeta \approx_{U \cap V} \rho \land \zeta \approx_{U \cap V} \eta$. Moreover, the disjointness of $U$ and $V$ implies that every behavior $\rho \in P$ can be synchronously combined with every behavior $\eta \in Q$. For this reason, we can construct $\zeta \in (P ∥ Q)$ for every $\rho \in P$ and $\eta \in Q$ as follows:

$$
\zeta(x) := \begin{cases} 
\rho(x) : & \text{if } x \in U \\
\eta(x) : & \text{if } x \in V 
\end{cases}
$$

\[\square\]

Note that the behavior $\zeta$ as constructed in the above proof is not the only one that is possible. Indeed, we can add silent actions in $\rho$ or $\eta$ to construct another behavior $\zeta'$ that is however not stretch equivalent to the $\zeta$. For this reason, these systems are not endochronous, but as the above corollary states, they are all isochronous.

The definition of isochronous compositions and the equivalences given in Lemma 7.54 can be easily generalized to more than two processes. As the aim is to replace a synchronous composition $P_1 ∥ ... ∥ P_n$ of processes $P_i$ over variables $V_i$ by their asynchronous composition $P_1 ∥_h ... ∥_h P_n$ (which is understood as $\text{flows}(P_1) ∥ ... ∥ \text{flows}(P_n)$), the generalization of Lemma 7.54 is as follows:
Lemma 7.57 (Isochronous Networks). For all processes \( P_1, \ldots, P_n \) over the variables \( V_1, \ldots, V_n \), respectively, the following propositions are equivalent:

1. \( \text{flows}(P_1) \parallel \ldots \parallel \text{flows}(P_n) = \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
2. \( \text{flows}(P_1) \parallel \ldots \parallel \text{flows}(P_n) = \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
3. \( (P_1 \parallel \ldots \parallel P_n) = \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
4. \( \text{flows}(P_1) \parallel \ldots \parallel \text{flows}(P_n) \subseteq \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
5. \( \text{flows}(P_1) \parallel \ldots \parallel \text{flows}(P_n) \subseteq \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
6. \( (P_1 \parallel \ldots \parallel P_n) \subseteq \text{flows}(P_1 \parallel \ldots \parallel P_n) \)
7. \( \forall \rho_1 \in P_1, \ldots, \forall \rho_n \in P_n. \)
   
   \[
   \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \rho_i \approx_{V_i \cap V_j} \rho_j \rightarrow \exists \zeta \in (P_1 \parallel \ldots \parallel P_n). \bigwedge_{i=1}^{n} \zeta \approx_{V_i} \rho_i
   \]
8. \( \forall \rho_1 \in P_1, \ldots, \forall \rho_n \in P_n. \)
   
   \[
   \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \rho_i \approx_{V_i \cap V_j} \rho_j \rightarrow \text{join}(\rho_1, \ldots, \rho_n) \in \text{flows}(P_1 \parallel \ldots \parallel P_n)
   \]

where \( \text{join}(\rho_1, \ldots, \rho_n)(x) := \begin{cases} 
\text{flows}(\rho_1)(x) & \text{if } x \in V_1 \\
\text{flows}(\rho_2)(x) & \text{if } x \in V_2 \\
\vdots & \vdots \\
\text{flows}(\rho_n)(x) & \text{if } x \in V_n
\end{cases} \)

The proofs of the equivalences listed in the above lemma are obtained analogously as in Lemma 7.54.

Note here that the parallel compositions \( \parallel \) and \( \parallel \) are both commutative and associative, and we also have \((P \parallel P) = P \) and \((P \parallel P) = P \) as well as \( P_1 \parallel (P_2 \parallel P_3) = (P_1 \parallel P_2) \parallel P_3 \) and \( P_1 \parallel (P_2 \parallel P_3) = (P_1 \parallel P_2) \parallel P_3 \). However, the definitions of isochrony of equivalent compositions like \((P_1 \parallel P_2 \parallel P_3) \parallel (P_1 \parallel P_2) \parallel P_3, \) and \((P_1 \parallel P_2) \parallel (P_1 \parallel P_3) \) lead to different demands according to the definition of isochrony. It is however not difficult to prove the following relationship:

Lemma 7.58. Given processes \( P_1, P_2, \) and \( P_3, \) over variables \( V_1, V_2, \) and \( V_3, \) respectively. Then, isochrony of \( P_1 \parallel P_2 \parallel P_3 \) implies isochrony of \( P_1 \parallel P_2, \) \( P_1 \parallel P_3, \) \( P_2 \parallel P_3, \) \( P_1 \parallel (P_2 \parallel P_3), \) and \((P_1 \parallel P_2) \parallel P_3, \)

Proof. Assume \( P_1 \parallel P_2 \parallel P_3 \) is isochronous, i.e. the following holds:

\[
\forall \rho_1 \in P_1, \forall \rho_2 \in P_2, \forall \rho_3 \in P_3.
\rho_1 \approx_{V_1 \cap V_2} \rho_2 \wedge \rho_1 \approx_{V_1 \cap V_3} \rho_3 \wedge \rho_2 \approx_{V_2 \cap V_3} \rho_3 \\
\rightarrow \exists \zeta \in (P_1 \parallel P_2 \parallel P_3). \zeta \approx_{V_1} \rho_1 \wedge \zeta \approx_{V_2} \rho_2 \wedge \zeta \approx_{V_3} \rho_3
\]

To prove that \( P_1 \parallel (P_2 \parallel P_3) \) is isochronous, we have to prove

\[
\forall \rho_1 \in P_1, \forall \eta \in (P_2 \parallel P_3).
\rho_1 \approx_{V_1 \cap (V_2 \cup V_3)} \eta \\
\rightarrow \exists \zeta \in (P_1 \parallel (P_2 \parallel P_3)). \zeta \approx_{V_1} \rho_1 \wedge \zeta \approx_{V_2 \cup V_3} \eta
\]
To this end, consider arbitrary behaviors $\rho_1 \in P_1$ and $\eta \in (P_2 \parallel P_3)$ that satisfy $\rho_1 \approx_{\eta} V_{f_1 \land f_2} \eta$, and therefore also $\rho_1 \approx_{\eta} V_{f_1 \land f_3} \eta$ and $\rho_1 \approx_{\eta} V_{f_1 \land f_2} \eta$. Moreover, since $\eta \in (P_2 \parallel P_3)$ holds, we know that $\eta_{f_2} \in P_2$ and $\eta_{f_3} \in P_3$ are true. For this reason, it follows by isochrony of $P_1 \parallel P_2 \parallel P_3$ that there is a behavior $\zeta \in (P_1 \parallel P_2 \parallel P_3)$ that satisfies the flow equivalences $\zeta \approx_{\eta} \rho_1$, $\zeta \approx_{\eta} \eta_{f_2}$, and $\zeta \approx_{\eta} \eta_{f_3}$. The latter two imply $\zeta \approx_{\eta} \eta$, and $\zeta \approx_{\eta} \eta$ by the projection lemma also $\zeta \approx_{\eta} (P_1 \parallel P_2 \parallel P_3)$. Finally, note that the behaviors of $(P_1 \parallel (P_2 \parallel P_3))$ and $P_1 \parallel P_2 \parallel P_3$ are the same.

Isochrony of $(P_1 \parallel P_2) \parallel P_3$ is obtained by commutativity of $\parallel$, so that we conclude that $P_3 \parallel (P_1 \parallel P_2)$ has to be isochronous which is true by the previous proof.

Finally, consider any process $P$ over some variables $V$. Isochrony of $P \parallel P$ means $\forall \rho \in P, \forall \eta \in P, \rho \approx_{\eta} \eta \to \exists \zeta \in P, \zeta \approx_{\eta} \rho \land \zeta \approx_{\eta} \eta$. It is easily seen that this is always true, since one can choose $\zeta := \rho$ as a witness. Thus, $P \parallel P$ is always isochronous. For this reason, we derive the isochrony of $P_1 \parallel P_2$ from $P_1 \parallel (P_2 \parallel P_2)$, and similarly we derive the isochrony of $P_1 \parallel P_3$ and $P_1 \parallel P_2 \parallel P_3$.

However, it is not the case that isochrony of all pairs $P_i \parallel P_j$ implies the isochrony of a network $P_1 \parallel \ldots \parallel P_n$. To see this, consider the processes $P_1 := CL_{cl}(\{\rho_1\})$, $P_2 := CL_{cl}(\{\rho_2\})$, and $P_3 := CL_{cl}(\{\rho_3\})$ with following behaviors $\rho_1$, $\rho_2$, and $\rho_3$:

\[
\rho_1 = \begin{cases} 
  x_1 & 0 \ 3 \ 6 \ldots \\
  x_2 & 1 \ 4 \ 7 \ldots \\
\end{cases} \quad \rho_2 = \begin{cases} 
  x_2 & 1 \ 4 \ 7 \ldots \\
  x_3 & 2 \ 5 \ 8 \ldots \\
\end{cases} \quad \rho_3 = \begin{cases} 
  x_1 & 0 \ 3 \ 6 \ldots \\
  x_2 & 2 \ 5 \ 8 \ldots \\
\end{cases} \\
\rho_{1,2} = \begin{cases} 
  x_1 & 0 \ 3 \ 6 \ldots \\
  x_2 & 1 \ 4 \ 7 \ldots \\
  x_3 & 2 \ 5 \ 8 \ldots \\
\end{cases} \quad \rho_{2,3} = \begin{cases} 
  x_2 & 1 \ 4 \ 7 \ldots \\
  x_3 & 2 \ 5 \ 8 \ldots \\
\end{cases} \quad \rho_{1,3} = \begin{cases} 
  x_1 & 0 \ 3 \ 6 \ldots \\
  x_2 & 1 \ 4 \ 7 \ldots \\
  x_3 & 2 \ 5 \ 8 \ldots \\
\end{cases}
\]

It is not difficult to see that $(P_1 \parallel P_2) = CL_{cl}(\{\rho_{1,2}\})$, $(P_1 \parallel P_3) = CL_{cl}(\{\rho_{1,3}\})$, and $(P_2 \parallel P_3) = CL_{cl}(\{\rho_{2,3}\})$ holds. However, there is no way to synchronously combine all three processes, so that $(P_1 \parallel P_2 \parallel P_3) = \{\}$ holds.

It is also not difficult to see that $\rho_1 \approx_{cl}^{\{x_2\}} \rho_2$, $\rho_1 \approx_{cl}^{\{x_1\}} \rho_3$, and $\rho_2 \approx_{cl}^{\{x_3\}} \rho_3$ holds, so that we also have $\rho_1 \approx_{\eta}^{\{x_2\}} \rho_2$, $\rho_1 \approx_{\eta}^{\{x_1\}} \rho_3$, and $\rho_2 \approx_{\eta}^{\{x_3\}} \rho_3$. Since $\rho_{1,2}$, $\rho_{1,3}$, and $\rho_{2,3}$, are synchronous combinations of pairs of behaviors $\rho_1$, $\rho_2$, and $\rho_3$, we conclude that the compositions $P_1 \parallel P_2$, $P_1 \parallel P_3$, and $P_2 \parallel P_3$ are all isochronous.

For the isochrony of $P_1 \parallel P_2 \parallel P_3$, however, we have to show that there is a combined behavior $\zeta \in (P_1 \parallel P_2 \parallel P_3)$ with $\zeta \approx_{\eta}^{\{x_1,x_2\}} \rho_1$, $\zeta \approx_{\eta}^{\{x_2,x_3\}} \rho_2$, and $\zeta \approx_{\eta}^{\{x_1,x_3\}} \rho_3$. We hope this helps you further.
and \( \zeta \approx_{\theta} \rho_3 \). However, as outlined above, \((P_1 \parallel P_2 \parallel P_3)\) is empty, so that the composition \(P_1 \parallel P_2 \parallel P_3\) is not isochronous.

Since \(P_1 \parallel P_2 \parallel P_3\) is the same as \((P_2 \parallel P_3)\) and also as \((P_1 \parallel P_2)\) \(\parallel P_3\), the pairwise isochrony of all \((P_i \parallel P_j)\) does also not imply the isochrony of \(P_1 \parallel (P_2 \parallel P_3)\) or \((P_1 \parallel P_2) \parallel P_3\).

We therefore conclude that Corollary 1 of [30, 31] is not correct, and looking at its proof, we find the following flaws (given in the above notation):

Let us call \(P \parallel Q\) strongly isochronous if the conditions (A) and (B) given in Lemma 7.54 imply condition (C). Assume \(P, Q_1\), and \(Q_2\) are processes over the variables \(U, V_1\), and \(V_2\), respectively. It is then claimed in [30, 31] that \(P \parallel (Q_1 \parallel Q_2)\) is strongly isochronous provided that both \(P \parallel Q_1\) and \(P \parallel Q_2\) are strongly isochronous. To this end, assume \(t \in \mathbb{N}\) is a position in synchronous behaviors \(\rho \in P\) and \(\eta \in (Q_1 \parallel Q_2)\) such that

(A.1) \(\exists x \in U \cap (V_1 \cup V_2). \rho(x)^{(t)} \neq \Box\)

(A.2) \(\exists x \in U \cap (V_1 \cup V_2). \eta(x)^{(t)} \neq \Box\)

(B) \(\forall x \in U \cap (V_1 \cup V_2). \rho(x)^{(t)} \neq \Box \land \eta(x)^{(t)} \neq \Box \rightarrow \rho(x)^{(t)} = \eta(x)^{(t)}\)

Hence, due to (A.2) \(\eta_{V_1}\), or \(\eta_{V_2}\) is not silent on all of its variables \(V_1\) and \(V_2\), respectively (since \(x \in U \cap (V_1 \cup V_2)\) is equivalent to \(x \in (U \cap V_1) \cup (U \cap V_2)\)). Without loss of generality, we can assume (A.2) \(\exists x \in U \cap V_1. \eta_{V_1}(x) \neq \Box\).

From assumption (B) above, we then derive that \(\rho\) and \(\eta_{V_1}\) agree on all shared present variables \(U \cap V_1\). The proof given in [30, 31] proceeds now with using isochrony of \(P \parallel Q_1\) to derive that \(\rho\) and \(\eta_{V_1}\) must agree on all shared variables \(U \cap V_1\). However, \(\rho\) can be silent on the shared variables \(U \cap V_1\), since (A.1) can be still be satisfied in this case when \(\rho\) is not silent on \(U \cap V_2\). Therefore the assumptions to make use of strong isochrony are not given, so that the conclusion made in [30, 31] is wrong. Hence, the proof is not complete/correct, and the example given above shows that the proof cannot be repaired.

We can now state the main theorem on the use of isochronous composition of endochronous systems:

### Theorem 7.59 (Using Endo-/Isochrony for Asynchronous Implementation)

Given a synchronous system \(P_1 \parallel \ldots \parallel P_n\) consisting of \(n\) components \(P_1, \ldots, P_n\) over variables \(V_1, \ldots, V_n\), respectively, so that the following holds

- Each \(P_i\) is endochronous, i.e., we can identify \(P_i\) with flows \((P_i)\) due to the existence of the resynchronization functions \(R_{P_i}\).
- The composition \(P_1 \parallel \ldots \parallel P_n\) is isochronous, i.e. for all behaviors \(\rho_1 \in P_1, \ldots, \rho_n \in P_n\) that can be asynchronously combined \((\rho_i \approx_{\rho_i \cap \rho_j} \rho_j)\) can also be synchronously combined.

Then, an asynchronous system with behaviors \(\text{flows}(P_1 \parallel \ldots \parallel P_n)\) can be obtained by adding wrappers around each \(P_i\) that implement the resynchronization function of \(P_i\), so that all communications including the internal communication between the components \(P_i\) is done asynchronously.
284  7 Synthesis of Asynchronous Systems

Fig. 7.19. Embedding an Endo-/Isochronous System in an Asynchronous Environment using Wrappers including the Asynchronous internal Communication

Proof. The proof is almost trivial by the previous results: Due to isochrony, \( \text{flows}(P_1 \parallel \ldots \parallel P_n) \) is the same as \( \text{flows}(P_1) \parallel \ldots \parallel \text{flows}(P_n) \), and due to endochrony of \( P_i \), \( \text{flows}(P_i) \) can be implemented by embedding the synchronous modules \( P_i \) in wrappers that implement their resynchronization functions \( R_{P_i} \), respectively.

\[ \therefore \]

The definitions of endochrony and isochrony given in this section have been taken essentially from [245, 246], while the definition of isochrony given in [157] is based on another definition of asynchronous composition which is as follows:

\[ P \parallel_a Q := \{ \xi | \exists \rho \in P. \exists \eta \in Q. \xi \approx_{U \setminus V}^d \rho \land \xi \approx_{V \setminus U}^d \eta \land \xi \approx_{U \cap V}^d \rho \land \xi \approx_{U \cap V}^d \eta \} \]

In contrast to \( (P \parallel Q) \), behaviors \( \xi \in (P \parallel_a Q) \) demand stretch equivalence on the ‘own’ variables \( U \setminus V \) and \( V \setminus U \), respectively, but only demand flow equivalences \( \xi \approx_{U \setminus V}^d \rho \land \xi \approx_{V \setminus U}^d \eta \) on the shared variables \( U \cap V \). As a consequence, \( P \parallel_a Q \) considers all values including \( \boxtimes \) for composing behaviors of \( P \) and \( Q \), while \( P \parallel_H Q \) only considers values different to \( \boxtimes \) for this purpose. In contrast to \( (P \parallel_H Q) \), the streams of the own variables are kept synchronized, and only the shared streams are desynchronized.

It is not difficult to see that \( (P \parallel Q) \subseteq (P \parallel_a Q) \) (since \( \xi \approx^d \rho \) implies \( \xi \approx_{U \setminus V}^d \rho \) and \( \xi \approx_{V \setminus U}^d \rho \), and thus \( \xi \approx_{U \cap V}^d \rho \) and that \( (P \parallel_a Q) \subseteq (P \parallel_H Q) \) holds (since \( \xi \approx_{U \setminus V}^d \rho \) implies \( \xi \approx_{U \setminus V}^d \rho \), and the latter with \( \xi \approx_{U \cap V}^d \rho \) implies according to Lemma 7.43 that \( \xi \approx_{U \setminus V}^d \rho \) holds).
For the converse, we can easily see that \((P \parallel Q) \not\subseteq (P \parallel_a Q)\) holds (since \(\otimes\) might occur in behaviors of \((P \parallel_a Q)\), but not in behaviors of \((P \parallel Q)\)), but we can prove the following result:

**Lemma 7.60.** For all processes \(P\) and \(Q\) over the variables \(U\) and \(V\), respectively, we have \((P \parallel Q) = \text{flows}(P \parallel_a Q)\).

**Proof.** The proof is as follows, where we note that for all \(\rho \in P\) and \(\eta \in Q\), the property \(\rho \approx_{U\setminus V} \eta\) is equivalent to the existence of a behavior \(\zeta\) with \(\zeta \approx_{U\setminus V} \rho\) and \(\zeta \approx_{V\setminus U} \eta\):

\[
\xi \in \text{flows}(P \parallel_a Q) \Leftrightarrow \exists \zeta \in (P \parallel_a Q). \xi = \text{flows}(\zeta) \\
\Leftrightarrow \exists \zeta, \exists \rho \in P, \exists \eta \in Q.
\]

\[
\xi = \text{flows}(\zeta) \land \zeta \approx_{U\setminus V} \rho \land \zeta \approx_{V\setminus U} \eta \land \zeta \approx_{U\cap V} \rho \land \zeta \approx_{U\cap V} \eta
\]

\[
\Leftrightarrow \exists \zeta, \exists \rho \in P, \exists \eta \in Q.
\]

\[
\xi_{\parallel U} = \text{flows}(\xi_{\parallel U}) \land \xi_{\parallel V} = \text{flows}(\xi_{\parallel V}) \land \\
\zeta \approx_{U\setminus V} \rho \land \zeta \approx_{V\setminus U} \eta \land \zeta \approx_{U\cap V} \rho \land \zeta \approx_{U\cap V} \eta
\]

\[
\Leftrightarrow \exists \rho \in P, \exists \eta \in Q.
\]

\[
\xi_{\parallel U} = \text{flows}(\rho) \land \xi_{\parallel V} = \text{flows}(\eta)
\]

\[
\Leftrightarrow \xi \in (P \parallel Q)
\]

\(\square\)

Thus, isochrony can be alternatively defined by means of \(\text{flows}(P \parallel_a Q)\) in that we demand that instead of \((P \parallel Q) \subseteq \text{flows}(P \parallel Q)\) the equivalent condition \((P \parallel_a Q) \subseteq \text{flows}(P \parallel Q)\) (or any other equivalent one according to Lemma 7.54). An advantage of the alternative definition might be that \(P \parallel_a Q\) emphasizes that the desynchronization only matters on the shared variables, while the other streams can be kept synchronized. Proofs become, however, more difficult.

For this reason, the definitions of endochrony and isochrony given in [157, 245, 246] are all the same, while the original ones given in [30, 31] differ from these: Endochrony as defined in [30, 31] is rather sequentiality of functions (which we distinguished from endochrony by the help of the Gustave function), and isochrony has been defined as the criterion mentioned in Lemma 7.55. Moreover, the original definitions given in [30, 31] refer to state transition systems while the other ones refer to traces, but this does not explain the differences between these definitions. We feel that the definitions made here which are in accordance to the newer references [157, 245, 246] match with the practical use outlined in Theorem 7.59.
7.3.3 Weak Endochrony

The definitions of endochrony and isochrony allows one to implement GALS from given synchronous systems that implement behaviors that are flow-equivalent to the given synchronous behaviors by means of the synchronous components and suitable wrapper modules. Although this is the fundamental theory for the synthesis of GALS from synchronous systems, the notions of endochrony and isochrony have some disadvantages. The most important disadvantage is the lack of compositionality, so that the generated GALS is in general not endochronous, so that we can in general not add a wrapper module to use it in a surrounding GALS.

For this reason, the approach has been modified in [211–213] where weak endochrony and weak isochrony have been defined as more liberalized forms of endochrony and isochrony. In contrast to endochrony, weak endochrony has the advantage of being compositional, so that the generated GALS can also be used as a weakly endochronous component for further implementations. A second advantage is that there are many systems that are weakly endochronous, but not endochronous, so that the GALS synthesis based on weak endochrony and weak isochrony can be applied to more synchronous systems as their stronger variants.

Weak endochrony extends the theory of Mazurkiewicz [91, 92] trace theory to synchronous systems: Behaviors of weakly endochronous systems are closed under commutation of independent actions where independence is defined by acting on disjoint sets of variables. Moreover, independent actions $\alpha_1, \alpha_2$ can be joined into a common action $\alpha_1 \sqcup \alpha_2$, and overlapping non-atomic reactions can be decomposed into atomic reactions that commute.

To start the formal definition of weakly endochronous systems, recall that behaviors $\xi \in \text{Bhv}(P)$ of a process $P$ over variables $V$ are variable assignments, i.e. functions of type $\xi : V \to (N \to D)$ that map variables $V$ of $P$ to streams of type $N \to D$. For each point of time $t \in N$, these variable assignments $\xi$ provide a reaction $\lambda v.\xi(v)(t)$ which is a function of type $V \to D$. Thus, by rearranging the function evaluation, variable assignments also provide a stream of reactions $\lambda t.\lambda v.\xi(v)(t)$ of type $N \to (V \to D)$.

For synchronous systems, a behavior can therefore also be adequately be represented as a sequence of reactions. Note again, that the value $\square$ has been introduced only for this purpose.

**Definition 7.61 (Reactions).** The reactions of a process $P$ over variables $V$ is defined as $\{ \rho : V \to D \mid \exists t \in N. \forall v \in V. \rho(v) = \xi(v)(t) \}$. The support $\text{support}(\alpha) := \{ v \in V \mid \alpha(v) \neq \square \}$ of a reaction $\alpha$ is the set of variables that are mapped to values different to $\square$.

Reactions $\alpha_1$ and $\alpha_2$ are called independent if $\text{support}(\alpha_1) \cap \text{support}(\alpha_2) = \{ \}$, and they are called joinable if both agree on the present variables, i.e.

$$\forall x \in V. \alpha_1(x) \neq \square \land \alpha_2(x) \neq \square \rightarrow \alpha_1(x) = \alpha_2(x)$$

For reactions $\alpha_1$ and $\alpha_2$, we moreover define the following reactions:
A reaction is called silent if it assigns all variables to $\square$, and it is called atomic if exactly one variable is mapped to a value different to $\square$.

Note that independent actions are also joinable (while the converse is false). It is easily seen that the following holds for the supports:

- $\text{support}(\alpha_1 \cap \alpha_2) = \text{support}(\alpha_1) \cap \text{support}(\alpha_2)$
- $\text{support}(\alpha_1 \cup \alpha_2) = \text{support}(\alpha_1) \cup \text{support}(\alpha_2)$
- $\text{support}(\alpha_1 \setminus \alpha_2) = \text{support}(\alpha_1) \setminus \text{support}(\alpha_2)$
- $\text{support}(\alpha_1 \Delta \alpha_2) = (\text{support}(\alpha_1) \setminus \text{support}(\alpha_2)) \cup (\text{support}(\alpha_2) \setminus \text{support}(\alpha_1))$

Finally, we make use of the concatenation $\rho \cdot \xi$ of a finite behavior $\rho$ with a possibly infinite behavior $\xi$ or a single reaction to define weakly endochronous systems:

**Definition 7.62 (Weakly Endochronous Systems [211–213]).** A process $P$ with input variables $V_{in}$ is weakly endochronous if the following holds:

**W1:** for all behaviors $\rho$ and reactions $\alpha$, $\beta$ with $\rho \cdot \alpha \in \text{Bhv}(P)$ and $\rho \cdot \beta \in \text{Bhv}(P)$, the following implication holds. $\alpha_{V_{in}} = \beta_{V_{in}} \Rightarrow \alpha = \beta$

**W2:** For independent reactions $\alpha_1, \alpha_2$ with both $\rho \cdot \alpha_1 \in \text{Bhv}(P)$ and $\rho \cdot \alpha_2 \in \text{Bhv}(P)$, we have

1. $\rho \cdot \alpha_1 \in \text{Bhv}(P)$ and $\rho \cdot \alpha_2 \in \text{Bhv}(P)$ implies $\rho \cdot (\alpha_1 \cup \alpha_2) \in \text{Bhv}(P)$
2. $\rho \cdot \alpha_1 \cdot \alpha_2 \in \text{Bhv}(P)$ implies $\rho \cdot \alpha_2 \in \text{Bhv}(P)$

**W3:** For joinable reactions $\alpha_1, \alpha_2$ with both $\rho \cdot \alpha_1 \in \text{Bhv}(P)$ and $\rho \cdot \alpha_2 \in \text{Bhv}(P)$, we also have $\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \setminus \alpha_2) \in \text{Bhv}(P)$ and $\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \in \text{Bhv}(P)$.

The important consequence for weakly endochronous systems is that they have the diamond property as proved in the following lemma (see also Figure 7.20). The diamond property is of essential importance for trace theory as outlined below:

**Lemma 7.63 (Full Diamond Lemma I).** Given that $P$ is weakly endochronous and for two reactions $\alpha_1$ and $\alpha_2$ with $\text{support}(\alpha_1) \cap \text{support}(\alpha_2) = \{\}$, we have $\rho \cdot \alpha_1 \in \text{Bhv}(P)$ and $\rho \cdot \alpha_2 \in \text{Bhv}(P)$, then $P$ also has the following behaviors:

- $\rho \cdot \alpha_1 \cdot \alpha_2 \in \text{Bhv}(P)$
- $\rho \cdot \alpha_2 \cdot \alpha_1 \in \text{Bhv}(P)$
- $\rho \cdot (\alpha_1 \cup \alpha_2) \in \text{Bhv}(P)$
The proofs are as follows: Applying W3 to (1) and (2) yields (5) \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \in \text{Bhv}(P) \). Analogously, we derive (5) \( \rho \cdot \alpha_2 \cdot \alpha_1 \in \text{Bhv}(P) \) from (2) and (3).

In addition to the additional three reactions mentioned in the lemma, one can also prove the \( P \) has furthermore the following additional behaviors:

- \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \in \text{Bhv}(P) \)
- \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \setminus \alpha_2) \cdot \alpha_2 \in \text{Bhv}(P) \)
- \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \cdot \alpha_1 \in \text{Bhv}(P) \)

The proofs are as follows: Applying W3 to (1) and (2) yields (5) \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \in \text{Bhv}(P) \) and (6) \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \in \text{Bhv}(P) \). In particular, we also have (7) \( \rho \cdot (\alpha_1 \cap \alpha_2) \in \text{Bhv}(P) \). Finally, we apply W3 to (3) and (7) and apply \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \cdot (\alpha_1 \setminus \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \in \text{Bhv}(P) \). Noting that \( (\alpha_1 \cap \alpha_2) \cap (\alpha_1 \cup \alpha_2) = (\alpha_1 \cap \alpha_2) \) and \( ((\alpha_1 \cup \alpha_2) \setminus (\alpha_1 \cap \alpha_2)) = \alpha_1 \Delta \alpha_2 \) proves \( \rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \in \text{Bhv}(P) \).

However, due to support(\( \alpha_1 \)) \( \cap \) support(\( \alpha_2 \)) = \{ \}, we have the following equations

- \( (\alpha_1 \cap \alpha_2)(v) = \Box \)
- \( \alpha_1 \setminus \alpha_2 = \alpha_1 \)
• \(\alpha_2 \setminus \alpha_1 = \alpha_2\)
• \(\alpha_1 \Delta \alpha_2 = \alpha_1 \cup \alpha_2\)

and therefore, the above additional behaviors boil down to the three mentioned in the lemma, with the exception that also silent actions are added. As we usually assume stretch closed systems, this does however also not add further information. For joinable reactions, however, the additional behaviors can also be proved and make sense. For this reason, we prove the following additional diamond lemma (see Figure 7.21):

\[
\rho \cdot \alpha_1 \cdot \alpha_2 \in \text{Bhv}(P) \\
\rho \cdot \alpha_2 \cdot \alpha_1 \in \text{Bhv}(P) \\
\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \Delta \alpha_2) \in \text{Bhv}(P) \\
\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \setminus \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \in \text{Bhv}(P) \\
\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \cdot (\alpha_1 \setminus \alpha_2) \in \text{Bhv}(P)
\]

**Proof.** Assume (1) \(\rho \cdot \alpha_1 \in \text{Bhv}(P)\) and (2) \(\rho \cdot \alpha_2 \in \text{Bhv}(P)\) holds for some joinable actions \(\alpha_1\) and \(\alpha_2\). By property W3 of weak endocrony, it then follows that also (3) \(\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_1 \setminus \alpha_2) \in \text{Bhv}(P)\) and (4) \(\rho \cdot (\alpha_1 \cap \alpha_2) \cdot (\alpha_2 \setminus \alpha_1) \in \text{Bhv}(P)\) holds. Clearly, \(\alpha_1 \setminus \alpha_2\) and \(\alpha_2 \setminus \alpha_1\) have disjoint support, so that Lemma 7.63 can applied using \(\rho' := \rho \cdot (\alpha_1 \cap \alpha_2)\), \(\alpha'_1 := \alpha_1 \setminus \alpha_2\), and \(\alpha'_2 := \alpha_2 \setminus \alpha_1\) to derive the desired behaviors. \(\Box\)
Note that $\rho \cdot (\alpha_1 \sqcup \alpha_2) \in \text{Bhv}(P)$ is not proved. A direct consequence of the diamond lemmas is the following theorem that is the key to the use of weak endochrony:

**Theorem 7.65 (Commutation Theorem).** If for a weakly endochronous system $P$, we have $\rho \cdot \alpha_1 \cdot \alpha_2 \in \text{Bhv}(P)$ for reactions $\alpha_1$, $\alpha_2$ with $\text{support}(\alpha_1) \cap \text{support}(\alpha_2) = \{\}$, then we also have $\rho \cdot \alpha_2 \cdot \alpha_1 \in \text{Bhv}(P)$.

**Proof.** By property W2.2, we derive $\rho \cdot \alpha_2 \in \text{Bhv}(P)$ from $\rho \cdot \alpha_1 \cdot \alpha_2 \in \text{Bhv}(P)$, so that the assumptions of the previous lemma are satisfied. Thus, we have $\rho \cdot \alpha_2 \cdot \alpha_1 \in \text{Bhv}(P)$. $\Box$

Due to the above theorem, we can define trace equivalence of synchronous behaviors that is in analogy to stretch equivalence in weak endochrony:

**Definition 7.66 (Trace Equivalence).** For synchronous behaviors of a process $P$, we define the trace equivalence relation $\approx_{\text{tr}}$ as the reflexive and transitive closure of the following relation, where $\rho_1$ and $\rho_2$ are sequences of reactions, and $\alpha_1$ and $\alpha_2$ are single reactions with disjoint support:

- $\rho_1 \cdot \alpha_1 \cdot \alpha_2 \cdot \rho_2 \approx_{\text{tr}} \rho_1 \cdot \alpha_2 \cdot \alpha_1 \cdot \rho_2$
- $\rho_1 \cdot \alpha_1 \cdot \alpha_2 \cdot \rho_2 \approx_{\text{tr}} \rho_1 \cdot (\alpha_1 \sqcup \alpha_2) \cdot \rho_2$
- $\rho_1 \cdot (\alpha_1 \sqcup \alpha_2) \cdot \rho_2 \approx_{\text{tr}} \rho_1 \cdot \alpha_1 \cdot \alpha_2 \cdot \rho_2$

Note that the silent action has a disjoint support to any other action. For this reason

**Lemma 7.67 (Trace Equivalence).** The relation $\approx_{\text{tr}}$ defined above is an equivalence relation, i.e.,

- $\forall \xi. \xi \approx_{\text{tr}} \xi$
- $\forall \xi_1, \xi_2. \xi_1 \approx_{\text{tr}} \xi_2 \Rightarrow \xi_2 \approx_{\text{tr}} \xi_1$
- $\forall \xi_1, \xi_2, \xi_3. \xi_1 \approx_{\text{tr}} \xi_2 \land \xi_2 \approx_{\text{tr}} \xi_3 \Rightarrow \xi_1 \approx_{\text{tr}} \xi_3$

Moreover, for weakly endochronous processes, stretch equivalence implies trace equivalence and trace equivalence in turn implies flow equivalence, i.e.,

- $\forall \xi_1, \xi_2. \xi_1 \approx_{\text{cl}} \xi_2 \Rightarrow \xi_2 \approx_{\text{tr}} \xi_1$
- $\forall \xi_1, \xi_2. \xi_1 \approx_{\text{tr}} \xi_2 \Rightarrow \xi_2 \approx_{\text{fl}} \xi_1$

**Proof.** Reflexivity and transitivity of $\approx_{\text{tr}}$ are given by definition of $\approx_{\text{tr}}$, and symmetry follows from the fact that the cases mentioned in the above definition are symmetric.

To prove that $\xi_1 \approx_{\text{cl}} \xi_2$ implies $\xi_2 \approx_{\text{tr}} \xi_1$, we show that adding an arbitrary number of silent actions in a behavior yields a trace equivalent behavior. To this end, consider a behavior $\rho \cdot \alpha \cdot \xi$ of a weakly endochronous process (see Figure 7.22). Since the process is stretch closed, also $\rho \cdot \tau$ is a behavior, where $\tau$ denotes the silent action that maps all variables to $\Box$. Since $\tau$ is independent of $\alpha$, we can apply Lemma 7.63 to obtain the upper diamond in Figure 7.22.
We can repeat this argument, so that we can add further diamonds as shown by the second one in Figure 7.22. As can be seen, we now have constructed behaviors \( \rho \cdot \tau \cdot \alpha \cdot \xi \), \( \rho \cdot \tau \cdot \alpha \cdot \xi \) and in general \( \rho \cdot \tau \cdot \ldots \tau \cdot \alpha \cdot \xi \) as well as \( \rho \cdot \alpha \cdot \tau \cdot \ldots \tau \cdot \xi \). Thus, we can add at any arbitrary point in a behavior a finite number of silent actions to stretch a given behavior. For this reason, all stretch equivalent behaviors are also trace equivalent.

Finally, observe that commuting independent actions, i.e., changing \( \rho_1 \cdot \alpha_1 \cdot \alpha_2 \cdot \rho_2 \) to \( \rho_1 \cdot \alpha_2 \cdot \alpha_1 \cdot \rho_2 \) does not modify the flows of these behaviors due to the independence of \( \alpha_1 \) and \( \alpha_2 \). The same holds for joining independent actions into one. Thus, \( \rho_1 \approx_{\text{tr}} \rho_2 \) implies \( \text{flows}(\rho_1) = \text{flows}(\rho_2) \) which means \( \rho_1 \approx_{\text{fl}} \rho_2 \).

Recall that endochronous systems have the ability to resynchronize input flows into the inputs of a set of reactions that the synchronous system can perform. This is no longer necessary for a weakly endochronous system since a weakly endochronous system can fire a part of a reaction so that reactions are partitioned into independent sub-actions. It is therefore sufficient for a wrapper module for a weakly endochronous system that it detects by inspecting the input flows whether one of these atomic actions can be fired. In contrast to endochronous systems, it is thereby not required that an entire synchronous action is executed.

For this reason, it is easily seen that stretch equivalence implies trace equivalence, and therefore

**Theorem 7.68 (Endochrony vs. Weak Endochrony).** Every endochronous system is weakly endochronous, but there are weakly endochronous systems that are not endochronous.

**Proof.** Clock equivalence \( \square \)
Moreover, it is proved in [213] that weakly endochronous processes $P$ and $Q$ are isochronous if they are non-blocking, i.e. that every finite behavior can be extended indefinitely.

Module cp2 as composed of two cp1 modules is a weakly endochronous system that is not endochronous. Also the composition of processes filter and merge as discussed for the motivation of isochrony forms a weakly endochronous system that is not endochronous.

**Theorem 7.69 (Compositionality of Weak Endochrony).** If $P$ and $Q$ are weakly endochronous systems, then also $P \parallel Q$ is weakly endochronous.

**Theorem 7.70 (Checking Endochrony/Isochrony [32]).** Assume that program $P$ has a clock hierarchy consisting of a single tree. Also assume that it is decomposed as $P = P_1 \parallel \ldots \parallel P_k$, and, for each $k$, the clock hierarchy of component $P_k$ is a subtree of the clock tree of $P$. Then the corresponding network is endo/isochronous.
7.3 Globally Asynchronous Locally Synchronous Systems

7.3.4 Elastic Synchronous Systems

**Definition 7.71 (Flow Invariance [157]).** The composition \( P \parallel Q \) is called flow invariant if for all \( \rho \in (P \parallel Q) \) and all \( \eta \in (P \parallel Q) \) the input flow equivalence \( \rho 
refl \eta \) implies general flow equivalence \( \rho \approx \eta \) (where \( \approx \nrefl \) refers to the input variables of \( P \parallel Q \)).

Thus, flow invariance is similar to latency insensitivity (see Definition 7.49).

**Lemma 7.72.** Assume \( P \) and \( Q \) are processes over the variables \( U \) and \( V \), respectively. If \( P, Q \) and \( P_{U \cap V} \parallel Q_{U \cap V} \) are endochronous, then \( P \parallel Q \) is flow invariant.

**Definition 7.73 (Patient Processes).** See [27]
Sanity Checks

As it is the case for all synchronous programming languages, Quartz has been given a formal semantics. This is not only necessary to reason about the correctness of synthesis algorithms, it is also the basis to verify that a program fulfill certain temporal and functional properties. The formal semantics allows us thereby to translate Quartz programs to formal methods that are used by formal verification procedures.

The properties that have to be checked can be classified into program-specific and general properties. General properties check for arithmetic overflows or overflows of array index expressions and are automatically generated by the compiler.

8.1 Surface-Depth Expansion of Statements

A major aspect of the formal semantics as well as for the synthesis procedures is the distinction between the surface and the depth parts of a statement, i.e., the entering behavior of a statement and the remaining behavior (where the control flow is resumed from somewhere inside the statement). The separation of surface and depth parts is the prerequisite to rename occurrences of local variables in the surfaces which is necessary to statically resolve reincarnation problems by the compiler.

The algorithms presented in the previous chapters are efficient and are able to handle all programs, so there is no need to modify them. However, in case that a formal verification or a simulation run reveals that a desired property does not hold, it remains to explain the sometimes difficult behavior to the programmer. To this end, causality problems are not a big problem since the constructiveness has already been described in an operational way that lends itself well for a step-by-step explanation of the program. However, schizophrenia problems and the potentially resulting reincarnated local variables impose again difficulties, since the same lines of code may be executed
several times with a different context. This makes it hard to illustrate the behavior at the source code level of the program.

To simplify these matters, we present in this section a program transformation that generates equivalent programs that do not suffer from schizophrenia: In every macro step, nearly all lines of the generated program are executed at most once. These programs are obtained by copying the surface parts of loops so that the different surfaces are presented to the programmer.

The program transformation of this section is based on the seminal approach presented in [233]. The main idea of that paper is to define for every statement $S$ corresponding surface and depth statements such that the surface statement contains all actions of $S$ that are executed when $S$ is entered, and the depth statement is the remaining part of $S$. Both statements are defined in [233] by a simple primitive recursion over the statements, and can be computed in time $O(|S|)$ and $O(|S|^2)$, respectively. The reason for the quadratic blow-up of the depth is again that some statements like sequences and loops generate copies of surface statements. It can be proved that $S$ is behaviorally equivalent to the sequence of the surface and depth statements.

To deal with reincarnated local variables, different ways can be followed: First of all, we can try to rename the local variables in the loop surfaces to avoid write conflicts when different scopes are executed in the same macro step. This has already been proposed in [233]. However, while it is simple to rename the variables in the surface statements, this is not sufficient:

- We additionally have to disable delayed assignments to the local variables that are enabled in the macro step where the scope is left.
- For local memorized variables, we might have to select the most recently executed surface to gather the value of that reincarnation for the reaction-to-absence in the macro step right after the new scope has been entered. This is necessary to determine the value of the local variable in case that there are no assignment in that macro step. To this end, we need all guarded actions to the local variables to determine the reaction-to-absence.
- The surface statement as defined in [233] does not cover the entering of the control flow. Instead, the surface only contains all the data actions that are executed at starting time of the statement, while the entire control flow has been retained in the depth statement. In particular, the entering of the control flow has still been retained in the depth statement. Therefore, the depth may still have to evaluate control flow expressions $\sigma$ (in if-statements, while-loops, and immediate preemptions) at starting time. The problem is that these expressions may still have schizophrenic occurrences so that we have to make sure that these conditions refer to the right context where they are executed. However, we can not simply rename these expression as can be done in the surfaces, since these occurrences of the conditions may still be schizophrenic. In [233], it has therefore been proposed to replace these conditions $\sigma$ with conditional instances of the
form $(\varphi) ? h(\sigma) : \sigma$, so that the condition $\varphi$ selects one of the instances $h(\sigma)$ and $\sigma$ similar to the selection of the most recently executed surface in the construction of the data flow transition relation in Section 4.5.3 and the data flow equations in Section 6.1. However, the overall approach becomes relatively complicated and produces sometimes large control flow expressions. Moreover, one has to pessimistically replace all control flow expressions that could be evaluated at starting time of a loop.

Looking at the symbolic SOS rules, it turns out that the enter predicate also belongs to the surface, which is indicated by its use of an incarnation level function. Moreover, the CompileSurface function of the hardware synthesis also computed the entering part of the control flow. Therefore, it seems to be obvious that the entering part of the control flow has to be covered by the surface statement. Covering the entering part of the control flow in the surface statement would therefore solve the last one of the above mentioned problems.

However, it would impose another problem: We can only determine in the surface statement which of the control flow locations should be active at the next point of time, but the surface statement does not contain these locations. Hence, we have to transfer this information somehow to the depth statement. Using local variables to store this information, e.g. by delayed assignments is problematic since preemption statements distinguish between control and data flow: while weak preemption statements retain the data actions of that macro step, they modify the control flow. Hence, we need new control flow statements to activate control flow locations outside the surface statement.

The statement that obviously has the required behavior is the goto statement has been introduced in Esterel by Tardieu in his PhD thesis [249] (see also [247, 250]). Intuitively, the semantics of $\text{goto}(\ell)$ is similar to the semantics of $\text{pause}$: the statement terminates the macro step, and the control flow should be resumed from control flow location $\ell$ at the next point of time. However, the $\text{goto}$ statement does not introduce a new control flow location, and instead refers to an existing one. Moreover, the depth behaviors of $\text{goto}$ and $\text{pause}$ are different (see the formal definitions in this section).

Clearly, the use of $\text{goto}$ can be problematic since it can destroy important invariants like the mutual exclusive activity of the substatements of if-statements and sequences. Moreover, using $\text{goto}$, one can send different activations through a statement similar to a pipelining of the statement. For this reason, we believe that its use is unsafe for programmers, and therefore, the statement should not be offered to the programmers.

However, using the goto statement, one can elegantly cover the entering of the control flow in the surface statement. Having determined a control flow location $\ell$ that has to be activated in the next macro step, we simply encode that activation by executing $\text{goto} \ \ell$ in the surface. Interestingly, this indicates that $\text{goto}$ can be viewed as the surface of $\text{pause}$. Therefore, there is no longer a need to evaluate control flow expressions at starting time in...
the depth statement. Instead, this is done in the surface statement, and the execution of the \texttt{goto} statements in the surface directly jump to the locations of the depth statement. As a consequence, the control flow conditions of the depth are now no longer schizophrenic.

Nevertheless, the renaming approach has further problems like the first and the second one we mentioned above that can not be solved without computing all guarded actions of the local variables and adding complex expressions to the program. As the intention is to keep the programs readable and to maintain the program structure as good as possible, we conclude that the renaming approach as used in the compile algorithms for hardware synthesis is not a reasonable way to go here. Clearly, we do not doubt that such an algorithm can be defined, but it will be a complicated algorithm comparable to the hardware synthesis that will modify the program’s expressions to a large extent.

We therefore propose another solution in this section: Instead of renaming the local variables in the surfaces, we make use of the \texttt{init} statement to express that a new scope of the variable \(x\) has to be created. This means that we remove all pending delayed actions on \(x\) as well as all immediate ones after this micro step. Hence, we do not remove the reincarnation, but it will be nevertheless more visible since now there are assignments above and below an \texttt{init} assignment. Therefore, the programmer can easily distinguish between the different actions of different reincarnations, which is our aim.

In this section, we first follow the approach of [233] in Section 8.1.1. In contrast to the original definitions of [233], we make use of the \texttt{init} statement instead of renamings in the depth statement. We show the advantages of the resulting program transformation by means of examples that also demonstrate problems that would appear with the renaming approach. In Section 8.1.2, we then present the alternative approach by introducing the \texttt{goto} statement. We also have to describe the semantics of this statement, although we do not add it to the Quartz language, since we believe that its use by the programmer is unsafe. However, its use by the compiler is safe and offers elegant program transformations as the one presented in Section 8.1.2.

8.1.1 Surface/Depth-Splitting without \texttt{goto}

In this section, we define the surface and depth statements of a given Quartz statement so that the surface statement contains all the data actions that are executed at starting time of the statement, while the depth statement contains the remaining behavior of the given statement.

We endow the computation of the surface with an incarnation level function \(\bar{h}\) to rename the expressions that are read and written at starting time of the statement. Although this is not used in the final program transformation, we provide this flexibility in case that other program transformations have to rename these expressions.
Definition 8.1 (Surface without goto). For every Quartz statement $S$ and every incarnation level function $\ell$, we recursively define the Quartz statement $\text{Surface}_h(S)$:

- $\text{Surface}_h(x=\tau) := h(x) = h(\tau)$
- $\text{Surface}_h(\text{next}(x)=\tau) := \text{next}(x) = h(\tau)$
- $\text{Surface}_h(\text{assume}(\sigma)) := \text{assume}(h(\sigma))$
- $\text{Surface}_h(\text{assert}(\sigma)) := \text{assert}(h(\sigma))$
- $\text{Surface}_h(\text{nothing}) := \text{nothing}$
- $\text{Surface}_h(\ell:\text{pause}) := \text{nothing}$
- $\text{Surface}_h(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) := \{\text{if}(h(\sigma)) \ \text{Surface}_h(S_1)\} \ \{\text{else} \ \text{Surface}_h(S_2)\}$
- $\text{Surface}_h(S_1 ; S_2) := \{\text{Surface}_h(S_1) : \text{if}(\text{inst}_h(S_1)) \ \text{Surface}_h(S_2)\}$
- $\text{Surface}_h(S_1 \ || \ S_2) := \text{Surface}_h(S_1) \ || \ \text{Surface}_h(S_2)$
- $\text{Surface}_h(\text{do} \ S \ \text{while}(\sigma)) := \text{Surface}_h(S)$
- $\text{Surface}_h(\{\alpha ; x ; S\}) := \text{Surface}_h(S)$
- $\text{Surface}_h(\text{during} \ S_1 \ \text{do} \ S_2) := \text{Surface}_h(S_1)$
- $\text{Surface}_h(\ell:\text{weak abort} \ S \ \text{when}(\sigma)) := \text{Surface}_h(S)$
- $\text{Surface}_h(\ell:\text{immediate abort} \ S \ \text{when}(\sigma)) := \text{Surface}_h(S)$
- $\text{Surface}_h(\ell:\text{weak immediate suspend} \ S \ \text{when}(\sigma)) := \text{Surface}_h(S)$
- $\text{Surface}_h(\ell:\text{immediate suspend} \ S \ \text{when}(\sigma)) := \text{Surface}_h(S)$

We can easily verify that the size of $\text{Surface}_h(S)$ grows linearly with the size of $S$, since every statement makes one subsequent call to its direct substatements plus some additional constructions that take constant time (provided that we already have constant time access to the already available control flow predicates). Hence, we can compute $\text{Surface}_h(S)$ with linear runtime and memory requirements.

Moreover, it is easily seen that for any Quartz statement $S$, the statement $\text{Surface}_h(S)$ is instantaneous. This is simply the case, since the surface statement does not contain any of the statements that have program locations $\ell$ like pause or the immediate suspension statements. Essentially, the surface statement only contains the surface actions of the statement, but not the information which control flow locations are entered by the statement. This can be seen in detail by the following facts:

Lemma 8.2 (Correctness of $\text{Surface}_h(S)$). The following facts hold for the statement $\text{Surface}_h(S)$ for any Quartz statement $S$:

- $\text{in} (\text{Surface}_h(S)) = \text{false}$
- $\text{inst}_h(\text{Surface}_h(S)) = \text{true}$
- $\text{enter}_h(\text{Surface}_h(S)) = \text{false}$
- $\text{term} (\text{Surface}_h(S)) = \text{false}$
stutter \( \text{Surface}_h(S) \) = true
move \( \text{Surface}_h(S) \) = false
move \( \text{Surface}_h(S) \) = false
\( \text{ActSurf}_h(\varphi; \text{Surface}_h(S)) = \text{ActSurf}_h(\varphi, S) \)
\( \text{ActDepth}(\text{Surface}_h(S)) = {} \)

The above properties are easily proved by a simple induction on the considered Quartz statement \( S \) by using the definitions.

It remains to define the depth statement \( \text{Depth}(S) \) for a statement \( S \). The statement \( \text{Depth}(S) \) differs from \( S \) only in that \( \text{Depth}(S) \) does not execute any data actions at starting time. The only differences to the original definitions of \cite{233} are that we make use of the \textit{init} statement as described below and that we consider additional statements.

\begin{definition}[Depth without goto] Given a Quartz statement \( S \), we define the Quartz statement \( \text{Depth}(S) \) recursively as follows:

\begin{itemize}
\item \( \text{Depth}(x=\tau) :\equiv \text{nothing} \)
\item \( \text{Depth}(\text{next}(x)=\tau) :\equiv \text{nothing} \)
\item \( \text{Depth}(\text{assume}(\sigma)) :\equiv \text{nothing} \)
\item \( \text{Depth}(\text{assert}(\sigma)) :\equiv \text{nothing} \)
\item \( \text{Depth}(\text{nothing}) :\equiv \text{nothing} \)
\item \( \text{Depth}(\ell:\text{pause}) :\equiv \ell:\text{pause} \)
\item \( \text{Depth}(\text{if}(\sigma) S_1 \text{ else } S_2) :\equiv \{ \text{if}(\sigma) \text{ Depth}(S_1) \text{ else } \text{Depth}(S_2) \} \)
\item \( \text{Depth}(\{S_1; S_2\}) :\equiv \begin{cases} \text{Depth}(S_1); \\
\text{if}(\in(S_1)) \text{Surface}_h(S_2); \\
\text{Depth}(S_2) \end{cases} \)
\item \( \text{Depth}(\{S_1 \parallel S_2\}) :\equiv \text{Depth}(S_1) \parallel \text{Depth}(S_2) \)
\item \( \text{Depth}(\{\alpha x; S\}) :\equiv \text{Depth}(S) \)
\item \( \text{Depth}(\text{during } S_1 \text{ do } S_2) :\equiv \text{during } \text{Depth}(S_1) \text{ do } S_2 \)
\item \( \text{Depth}(\text{abort } S \text{ when}(\sigma)) :\equiv \text{abort } \text{Depth}(S) \text{ when}(\sigma) \)
\item \( \text{Depth}(\text{weak abort } S \text{ when}(\sigma)) :\equiv \text{weak abort } \text{Depth}(S) \text{ when}(\sigma) \)
\item \( \text{Depth}(\text{suspend } S \text{ when}(\sigma)) :\equiv \text{suspend } \text{Depth}(S) \text{ when}(\sigma) \)
\item \( \text{Depth}(\text{weak suspend } S \text{ when}(\sigma)) :\equiv \text{weak suspend } \text{Depth}(S) \text{ when}(\sigma) \)
\item \( \text{Depth}(\text{immediate abort } S \text{ when}(\sigma)) :\equiv \{ \text{if}(\neg \sigma) \text{ abort } \text{Depth}(S) \text{ when}(\sigma) \} \)
\item \( \text{Depth}(\text{weak immediate abort } S \text{ when}(\sigma)) :\equiv \{ \text{if}(\neg \sigma) \text{ weak abort } \text{Depth}(S) \text{ when}(\sigma) \} \)
\end{itemize}
\end{definition}
8.1 Surface-Depth Expansion of Statements

- Depth (\ell:immediate suspend S when(\sigma))
  \[
  \begin{align*}
  & \text{if}(\sigma) \{ \\
  & \quad \ell:pause; \\
  & \quad \text{while}(\sigma); \\
  & \quad \text{Surface}_{\nu}(S) \\
  & \} \\
  & \text{suspend \ Depth}(S \ \text{when}(\sigma))
  \end{align*}
  \]

- Depth (\ell:weak immediate suspend S when(\sigma))
  \[
  \begin{align*}
  & \text{if}(\sigma) \{ \\
  & \quad \ell:pause; \\
  & \quad \text{Surface}_{\nu}(S) \\
  & \} \ \text{while}(\sigma); \\
  & \text{weak suspend \ Depth}(S \ \text{when}(\sigma))
  \end{align*}
  \]

Note that the definition of Depth (\Gamma) S removes all actions, but leads to subsequent calls to Surface_{\nu}(S) that maintain the actions. We discuss some of the above cases in more detail: The definition for the atomic statements should be clear (note that instantaneous statements have no depth actions). The definitions of the if-statement and the parallel statement are trivial.

The depth of sequences is a challenging definition: To understand the definition of Depth (S_1; S_2), consider the cases where S_1 is or is not instantaneous for the current input. If S_1 is instantaneous, then the depth of S_1; S_2 is the depth of S_2, otherwise, it is equivalent to Depth (\Gamma) S_1; S_2. At a first glance, we would therefore like to define the depth of S_1; S_2 as follows:

\[
\begin{align*}
\text{if}(\text{inst}_{\nu}(S_1)) & \ \text{Depth}(S_2) \\
\text{else} & \ \{\text{Depth}(S_1);S_2\}
\end{align*}
\]

Clearly, this definition can be improved by factoring out the depth of S_2 which is executed in both cases of the if-statement:

\[
\begin{align*}
\text{if}(\text{inst}_{\nu}(S_1)) & \ \text{nothing;} \\
\text{else} & \ \{\text{Depth}(S_1);\text{Surface}_{\nu}(S_2)\} \\
\text{Depth}(S_2)
\end{align*}
\]

We may now further observe that Depth (S_1) is behaviorally equivalent to nothing if S_1 is instantaneous. Hence, we can also rewrite the statement once more:

\[
\begin{align*}
\text{Depth}(S_1); \\
\text{if}(!\text{inst}_{\nu}(S_1)) & \ \text{Surface}_{\nu}(S_2); \\
\text{Depth}(S_2)
\end{align*}
\]

The version used in the above definition is yet another alternative that does not make use of the instantaneous-predicate so that there is no need for an incarnation level function \nu.
The symbolic SOS rules and the hardware synthesis updates the incarnation level function \( h \) whenever a loop is encountered. In this case, we increment the incarnation levels of all variables that are locally declared in the loop body. In an analogous way, we use the init statement to perform this task when we compute the depths of loops. Hence, the surface of the loop body that is potentially executed after the depth is executed in new scopes of the local variables that are declared inside the loop body.

The syntax of the init statement used for the definition above requires that the statement will always refer to another statement similar as a local declaration \( \{ \alpha x; S \} \) also refers to another statement \( S \). However, the init statement is placed in sequence after the statement in its scope, since it is assumed that the variables are already declared. The meaning of init is to redeclare the variables, i.e., to disable pending delayed assignments to the mentioned variables, and to distinguish between the immediate assignments before and after the init statement. The SOS transition rules of the init statement is as follows, where \( \text{Inc}(h', V) \) increments the incarnation levels of the variables \( V \):

\[
\langle E, h, S \rangle \rightarrow \langle h', S', D, \text{false} \rangle
\]

\[
\langle E, h, \{S; \text{init}(V);\} \rangle \rightarrow \langle h', \{S'; \text{init}(V);\}, D, \text{false} \rangle
\]

\[
\langle E, h, S \rangle \rightarrow \langle h', \text{nothing}, D, \text{true} \rangle
\]

\[
\langle E, h, \{S; \text{init}(V);\} \rightarrow \langle \text{Inc}(h', V), \text{nothing}, \{(y, v) \in D \mid x \not\in V\}, \text{true} \rangle
\]

Hence, init is similar to a local declaration in that it refreshes the local variables when necessary. In contrast to the local declaration, this is however done after the scope is left (or before a new scope is re-entered).

Note further that immediate preemption statements are eliminated by the surface-depth transformation. Note that we could alternatively use while-loops for the transformation of immediate suspension statements, but used the do-loops to only use core statements.

The crucial point is that the control flow of \( \text{Depth}(S) \) and \( S \) is the same provided that the control flow is already somewhere inside the statement \( S \). That is, the terminate and move predicate must be the same as well as the depth actions. This is stated by the following lemma:

**Lemma 8.4 (Correctness of \( \text{Depth}(S) \)).** Given a Quartz statement \( S \), the following facts are valid for \( \text{Depth}(S) \):

- \( \text{in}(\text{Depth}(S)) = \text{in}(S) \)
- \( \text{inst}_h(\text{Depth}(S)) = \text{inst}_h(S) \)
- \( \text{enter}_h(\text{Depth}(S)) = \text{enter}_h(S) \)
- \( \text{term}(\text{Depth}(S)) = \text{term}(S) \)
- \( \text{stutter}(\text{Depth}(S)) = \text{stutter}(S) \)
- \( \text{move}(\text{Depth}(S)) = \text{move}(S) \)

\(^1\) Note that we can talk about ‘before’ and ‘after’ in this setting.
• ActSurf\(_h(\varphi, \text{Depth}(S)) = \{\}\)
• ActDepth(\text{Depth}(S)) = \text{ActDepth}(S)

Again, each equation is easily proved by an induction over the set of Quartz statements \(S\). Combining the previous two lemmas, we can now conclude that the control flow of \(S\) and \(\text{GSurface}_{h_0}(S)\); \text{Depth}(S) is the same:

**Lemma 8.5 (Surface/Depth-Splitting).** Given a Quartz statement \(S\), the following equations are valid for all statements \(S\) and all functions \(h\):

- in(\{Surface\(_{h_0}(S)\); Depth(S)\}) = in(S)
- inst\(_h\)\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{inst}_h(S)
- enter\(_h\)\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{enter}_h(S)
- term\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{term}(S)
- stutter\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{stutter}(S)
- move\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{move}(S)
- ActSurf\(_h(\varphi, \{Surface\(_{h_0}(S); Depth(S)\}\}) = \text{ActSurf}_h(\varphi, S)
- ActDepth\{Surface\(_{h_0}(S); Depth(S)\}\} = \text{ActDepth}(S)

Hence, the statements \(S\) and \{Surface\(_{h_0}(S); Depth(S)\}\} are behaviorally equivalent. The proofs of the above equations can be directly obtained from the definition of the control flow predicates for sequences and the previous two lemmas. Concerning the complexity of the surface/depth splitting, we prove the following lemma:

**Lemma 8.6 (Size of Surface and Depth).** For every Quartz statement \(S\), the statements \(\text{Surface}_{h_0}(S)\) and \(\text{Depth}(S)\) have sizes \(O(|S|)\) and \(O(|S|^2)\), respectively (provided that the control flow predicates have been pre-computed).

We can use the statements \(\text{Surface}_{h_0}(S)\) and \(\text{Depth}(S)\) to replace \(S\) with the sequence \{Surface\(_{h_0}(S); Depth(S)\}\}. This is convenient to make the surface more explicit to avoid schizophrenic surfaces. The following list summarizes how the compound statements are thereby transformed in that a line marks the border of surface and depth:

- \(\text{if}(\sigma)\; S_1\; \text{else}\; S_2 = \{\text{if}(\sigma)\; \text{Surface}_{h_0}(S_1)\; \text{else}\; \text{Surface}_{h_0}(S_2)\} \text{if}(\sigma)\; \text{Depth}(S_1)\; \text{else}\; \text{Depth}(S_2)\} \)
- \(\{S_1; S_2\} = \left\{\begin{array}{l}
\text{Surface}_{h_0}(S_1) \\
\text{if}(\text{inst}_{h_0}(S_1))\; \text{Surface}_{h_0}(S_2)
\end{array}\right\}\text{Depth}(S_1)\;\text{Depth}(S_2)\)
- \(\{S_1 \parallel S_2\} = \left\{\begin{array}{l}
\{\text{Surface}_{h_0}(S_1) \parallel \text{Surface}_{h_0}(S_2)\} \\
\{\text{Depth}(S_1) \parallel \text{Depth}(S_2)\}
\end{array}\right\}\)
- \(\text{do}\; S\; \text{while}(\sigma)\equiv\left\{\begin{array}{l}
\{\text{Depth}(S); \text{init(LocVar}(S));\} \\
\text{if}(\sigma)\; \text{Surface}_{h_0}(S)\}
\end{array}\right\}\text{while}(\sigma)\)
304 8 Sanity Checks

- **abort** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{abort Depth (S)} \text{ when}(\sigma)
\end{cases}
\]

- **weak abort** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{weak abort Depth (S) when}(\sigma)
\end{cases}
\]

- **suspend** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{suspend Depth (S) when}(\sigma)
\end{cases}
\]

- **weak suspend** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{weak suspend Depth (S) when}(\sigma)
\end{cases}
\]

- **immediate abort** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{if}(!\sigma) \text{ abort Depth (S) when}(\sigma)
\end{cases}
\]

- **weak immediate abort** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{if}(!\sigma) \text{ weak abort Depth (S) when}(\sigma)
\end{cases}
\]

- **$\ell$ː immediate suspend** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{if}(!\sigma) \\
\text{if}(\sigma) \{
\text{do}
\text{Surface}_h (S) \\
\text{while}(\sigma) \\
\text{suspend Depth (S) when}(\sigma)
\}
\end{cases}
\]

- **$\ell$ː weak immediate suspend** $S$ when($\sigma$) ≡ \[
\begin{cases}
\text{Surface}_h (S) \\
\text{if}(\sigma) \{
\text{do}
\text{Surface}_h (S) \\
\text{while}(\sigma) \\
\text{weak suspend Depth (S) when}(\sigma)
\}
\end{cases}
\]

To conclude this section, let us consider how the surface/depth splitting can be used to make effects of schizophrenic statements clearer. Figure 8.1 shows one of the simplest programs with a schizophrenic local declaration. The expanded version of module $R_0$ is shown on the right hand side of Figure 8.1 and makes clear that after the first point of time, the module will emit both outputs; one because of the emission of the local variable $x$ in the depth, and the other because of the absence of emission of the reincarnation $x$. Note that the code below the pause statement is partitioned into two parts that refer to the old and the new scope of the local variable $x$. 
Next, consider the module \(R_1\) shown on the left hand side of Figure 8.2. The right hand side of the figure shows the module after applying the surface/depth splitting. As can be seen, two copies of the surface of the loop's body statement have been generated and in the second one (that is executed after re-entering the loop) is separated from the code executed in the depth by the \texttt{init} statement.
Module R1 would impose a major problem if we would like to rename the local variables in the surfaces. If we would do so, we would have to replace the occurrences of \( x \) in the depth by a conditional expression of the form \((\varphi \pi 1:x)\). While this can be achieved somehow for the first if-statement, it should not be done for the second if-statement which is however hard to determine by the compiler. Fortunately, the use of \texttt{init} makes it not necessary to rename the local variables.

![Module R2 and ExpandR2](image)

**Fig. 8.3.** Curing Schizophrenia Example R2.

Module R2 shown in Figure \( 8.8 \) is even more complicated: Note that the loop is never instantaneous, since the control flow must stop at least at location \( \ell_2 \). There is a difficult situation where \texttt{emit b} is three times executed in the expanded version, but only two times in the original one: Assume that the control flow is currently at location \( \ell_1 \) and not at \( \ell_2 \) (which is reached at starting time when \( k_1 \) is absent) and assume that \( k_1 \) present. Therefore the abortion takes place. As it is a weak abortion, we execute \texttt{emit b} after the if-statement. In the renewed execution of the loop body, the presence of \( k_1 \) is ignored by the abort statement, since it is not an immediate abort statement. The control flow will however not rest at \( \ell_1 \) since \( k_1 \) is present. Instead, we once more execute \texttt{emit b} after the conditional and the control will then finally rest at \( \ell_2 \).

Finally, consider how the module shown in Figure \( 8.4 \) is transformed by the surface/depth expansion. Depending on the inputs \( a \) and \( b \) the local variables may have one or two reincarnations, which is better shown in the expanded version given in Figure \( 8.5 \).
module MultipleReincar(event a,b,&y00,&y01,&y11, event int z) {
loop {
    event x1;
    weak abort
    loop {
        event x2;
        weak abort
        loop {
            switch
            case(!x1&!x2) do {emit y00; ℓ0:pause; z=0;}
            case(!x1& x2) do {emit y01; ℓ1:pause; z=1;}
            case( x1&!x2) do {emit y10; ℓ2:pause; z=2;}
            case( x1& x2) do {emit y11; ℓ3:pause; z=3;}
            else nothing;
                emit x1;
                emit x2;
        }
        when(b)
    }
    when(a)
}
}

Fig. 8.4. Multiple Reincarnations.

At starting time, the module emits y00, since neither x1 nor x2 holds. For this reason, the location ℓ0 is entered. The next macro step depends on the values of a and b:

- If neither a nor b holds, then the module executes z=0, then it emits both x1 and x2, so that also y11 is emitted. The loop iterates, and the new control flow location is ℓ3.
- If a is false, but b is true, then the module also executes z=0, emits both x1 and x2, so that also y11 is emitted. The loop iterates, but now the weak abortion terminates it. For this reason, the switch statement after the abortion statement with condition b is executed so that the module also emits y10 before entering location ℓ3.
- If a is true, but b is false, then the module executes z=0, emits both x1 and x2, so that also y11 is emitted. The loop iterates, but the weak abortion of condition a terminates it. For this reason, the switch statement after the abortion statement with condition a is executed so that the module also emits y00 before entering location ℓ0.
- Finally, if both a and b are true, then the module executes z=0, emits both x1 and x2, so that y11 is emitted. The loop iterates, but the weak abortion of condition b terminates it. Then, the switch statement after the abortion statement with condition b is executed so that the module emits y10. Then,
module ExpandMultipleReincar(event a,b,&y00,&y01,&y10,&y11, event int z) {
    event x1,x2;
    switch
        case(!x1&!x2) do {emit y00;}
        case(!x1& x2) do {emit y01;}
        case( x1&!x2) do {emit y10;}
        case( x1& x2) do {emit y11;}
    else nothing;
    loop {{
        weak abort {
            loop {
                weak abort {
                    loop {
                        switch
                            case(!x1&!x2) do {
                                ℓ0:pause; z=0;
                            }
                            case(!x1& x2) do {
                                ℓ1:pause; z=1;
                            }
                            case( x1&!x2) do {
                                ℓ2:pause; z=2;
                            }
                            case( x1& x2) do {
                                ℓ3:pause; z=3;
                            }
                        else nothing;
                        emit x1;
                        emit x2;
                        switch
                            case(!x1&!x2) do {emit y00;}
                            case(!x1& x2) do {emit y01;}
                            case( x1&!x2) do {emit y10;}
                            case( x1& x2) do {emit y11;}
                        else nothing;
                    }
                } when(b);
                init {x2};
            }
        } when(a);
        init {x1,x2};
    };
    switch
        case(!x1&!x2) do {emit y00;}
        case(!x1& x2) do {emit y01;}
        case( x1&!x2) do {emit y10;}
        case( x1& x2) do {emit y11;}
    else nothing;
} }

Fig. 8.5. Surface/Depth Transformation of MultipleReincar
the middle loop iterates, but is again aborted by abortion statement with condition \(a\). Hence, the module furthermore emits \(y00\) before entering location \(\ell_0\).

In locations \(\ell_1\) and \(\ell_3\), we observe the same behavior, except that other assignments to \(z\) are executed. Hence, location \(\ell_2\) is not reachable. As can be seen, the behavior of the module can be much better explained with the expanded version.

### 8.1.2 Surface/Depth-Splitting with \(\texttt{goto}\)

It may be disturbing that the surface statement of the previous section does not cover the entering behavior of the control flow, although this behavior belongs to the starting behavior of the statement. Note that the entering part has been viewed as part of the surface in the symbolic SOS rules as well as in the hardware synthesis. In this section, we therefore discuss how the entering behavior can be covered by the surface statement.

The definition of this surface statement requires the use of \(\texttt{goto}\) statements as proposed by Tardieu \[247, 249, 250\]. For this reason, we first have to define the syntax and semantics of this new statement. Concerning the syntax, the statement \(\texttt{goto } \ell\) refers to an existing control flow location \(\ell\). The semantics can not be easily given in terms of the SOS rules that work with residual statements \[249\]. Instead, it requires the use of the haltset encoding (also called the state behavioral semantics) of the residual statements. Intuitively, executing \(\texttt{goto } \ell\) behaves like \(\ell: \text{pause}\), i.e., no data actions are executed, and the macro step is terminated. The next macro step, however, starts then at control flow location \(\ell\).

As an alternative to the haltset encoding, the semantics of \(\texttt{goto } \ell\) can also be easily described with the symbolic SOS rules. We therefore give the following definition of the semantics of \(\texttt{goto } \ell\):

**Definition 8.7 (Semantics of \(\texttt{goto } \ell\)).** The semantics of the statement \(\texttt{goto } \ell\) is formally defined using the symbolic SOS rules as follows:

- \(\text{in}(\texttt{goto } \ell) := \text{false}\)
- \(\text{inst}_b(\texttt{goto } \ell) := \text{false}\)
- \(\text{enter}_b(\texttt{goto } \ell) := \text{next}(\ell)\)
- \(\text{term}(\texttt{goto } \ell) := \text{false}\)
- \(\text{move}(\texttt{goto } \ell) := \text{false}\)
- \(\text{ActSurf}_b(\varphi, \texttt{goto } \ell) := \{\}\)
- \(\text{ActDepth}(\texttt{goto } \ell) := \{\}\)

Comparing the semantics of \(\texttt{goto } \ell\) with \(\ell: \text{pause}\), it is seen that the semantics concerning the starting behavior like instantaneity, entering, and the surface actions are the same for \(\texttt{goto } \ell\) and \(\ell: \text{pause}\). However, the behavior in the depth like termination, internal moves is different.
Since goto statements are viewed as control flow statements, they can be preempted by suspension and abortion statements. Using the goto statement, the surface statement can now be defined as follows:

**Definition 8.8 (Surface with Goto).** For every Quartz statement \( S \) and every incarnation level function \( h \), we recursively define the Quartz statement \( \text{GSurface}_h(S) \) as follows:

- \( \text{GSurface}_h(x = \tau) \equiv h(x) = h(\tau) \)
- \( \text{GSurface}_h(\text{next}(x) = \tau) \equiv \text{next}(x) = h(\tau) \)
- \( \text{GSurface}_h(\text{assume}(\sigma)) \equiv \text{assume}(h(\sigma)) \)
- \( \text{GSurface}_h(\text{assert}(\sigma)) \equiv \text{assert}(h(\sigma)) \)
- \( \text{GSurface}_h(\text{nothing}) \equiv \text{nothing} \)
- \( \text{GSurface}_h(\ell: \text{pause}) \equiv \text{goto} \ell \)
- \( \text{GSurface}_h(\text{if}(\sigma) \ S_1 \ \text{else} \ S_2) \equiv \left\{ \begin{array}{ll} \text{if}(h(\sigma)) & \text{GSurface}_h(S_1) \\ \text{else} & \text{GSurface}_h(S_2) \end{array} \right\} \)
- \( \text{GSurface}_h(S_1; S_2) \equiv \left\{ \begin{array}{ll} \text{GSurface}_h(S_1) \\ \text{if}(\text{inst}_h(S_1)) & \text{GSurface}_h(S_2) \end{array} \right\} \)
- \( \text{GSurface}_h(S_1 \| S_2) \equiv \text{GSurface}_h(S_1) \| \text{GSurface}_h(S_2) \)
- \( \text{GSurface}_h(\text{do} \ S \ \text{while}(\sigma)) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\{\alpha \ x; \ S\}) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\text{during} \ S_1 \ \text{do} \ S_2) \equiv \text{GSurface}_h(S_1) \)
- \( \text{GSurface}_h(\text{abort} \ S \ \text{when}(\sigma)) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\text{suspend} \ S \ \text{when}(\sigma)) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\text{weak abort} \ S \ \text{when}(\sigma)) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\text{weak suspend} \ S \ \text{when}(\sigma)) \equiv \text{GSurface}_h(S) \)
- \( \text{GSurface}_h(\text{weak immediate abort} \ S \ \text{when}(\sigma)) \)
- \( \equiv \text{weak immediate abort} \text{Surface}_h(S) \ \text{when}(\sigma) \)
- \( \text{GSurface}_h(\text{immediate abort} \ S \ \text{when}(\sigma)) \)
- \( \equiv \text{if}(\text{th}(\sigma)) \text{Surface}_h(S) \)
- \( \text{GSurface}_h(\ell: \text{weak immediate suspend} \ S \ \text{when}(\sigma)) \)
- \( \equiv \text{Surface}_h(S); \ \text{if}(h(\sigma)) \ \text{goto} \ell \)
- \( \equiv \left\{ \begin{array}{ll} \text{weak immediate abort} \text{Surface}_h(S) \ \text{when}(h(\sigma)); \end{array} \right\} \)
- \( \text{GSurface}_h(\ell: \text{immediate suspend} \ S \ \text{when}(\sigma)) \)
- \( \equiv \text{if}(h(\sigma)) \ \text{goto} \ell \ \text{else} \text{Surface}_h(S) \)

There are some changes compared to \( \text{Surface}_h(S) \): We defined the surface of \( \ell: \text{pause} \) as \text{goto} \( \ell \), instead of nothing. Moreover, the immediate abortion statements must now also abort the entering moves of the control flow that are now included in the surface. For the strong abortion, we can handle this as shown above by an if-statement, but we could alternatively also use the equivalent statement \text{immediate abort} \text{Surface}_h(S) \ \text{when}(\sigma).

The definition of the surface of weak immediate suspension is tricky. We have provided two solutions: The first solution we used above makes use of the surface statement without \text{goto}, since we have to execute the data actions
of the surface in any case. The entering of the control flow depends however, on the suspension condition $\sigma$: If $\sigma$ holds, we enter the location $\ell$, otherwise we enter the statement $S$.

The second solution makes use of a weak-immediate abortion to kill the goto statements of the surface in case that the suspension immediately takes place. This makes also clear, that we can derive the surface statement without goto from the one above due to the following simple equivalence:

$$\text{Surface}_h(S) = \text{weak immediate abort } G\text{Surface}_h(S) \text{ when(true)}$$

There is another ‘solution’ that has however some drawbacks: One can argue that we could also use $\ell:\text{weak immediate suspend } G\text{Surface}_h(S) \text{ when(}\sigma\text{)}$ for the definition of $G\text{Surface}_h(\ell:\text{weak immediate suspend } S \text{ when(}\sigma\text{)})$.

However, then the surface statement would have control flow locations, which is something we want to avoid (see the next Lemma).

The size of $G\text{Surface}_h(S)$ grows linearly with the size of $S$ like $\text{Surface}_h(S)$. However, it is no longer the case that the statement $G\text{Surface}_h(S)$ is always instantaneous, since non-instantaneous goto statements are introduced.

**Lemma 8.9 (Correctness of $G\text{Surface}_h(S)$).** The following facts hold for the statement $G\text{Surface}_h(S)$ for any Quartz statement $S$:

- $\text{in}(G\text{Surface}_h(S)) = \text{false}$
- $\text{inst}_h(G\text{Surface}_h(S)) = \text{inst}_h(S)$
- $\text{enter}_h(G\text{Surface}_h(S)) = \text{enter}_h(S)$
- $\text{term}(G\text{Surface}_h(S)) = \text{false}$
- $\text{stutter}(G\text{Surface}_h(S)) = \text{true}$
- $\text{move}(G\text{Surface}_h(S)) = \text{false}$
- $\text{move}(G\text{Surface}_h(S)) = \text{false}$
- $\text{ActSurf}_h(\varphi,G\text{Surface}_h(S)) = \text{ActSurf}_h(\varphi,S)$
- $\text{ActDepth}(G\text{Surface}_h(S)) = \{\}$

The above properties are easily proved by a simple induction on the considered Quartz statement $S$ by using the definitions.

There is no need to define a corresponding depth statement for our new surface statement. Recall that our aim is to cure schizophrenia, and to this end, we have to copy the surfaces that are overlapped by loops. For this reason, we can simply perform the following simple transformation:

**Definition 8.10 (Surface/Depth Expansion with Goto).** Given a Quartz statement $S$, we replace all loops in a bottom-up traversal on the syntax tree as follows:

$$\text{Expand(do } S \text{ while(}\sigma\text{))} := \begin{cases} \{S; \text{init(LocVar}(S));\} & \text{if(}\sigma\text{) } \text{GSurface}(S); \\ \end{cases}$$
Using the goto statement, we can eliminate loops as shown above. The obtained statement has again a size $O(|S|^2)$. Clearly, we could also concentrate on loops when we follow the surface/depth expansion of the previous section. However, as we retain the loops in that expansion, we have to define a suitable depth statement.

Let us revisit the examples we already considered in the previous section. Figure 8.6 shows the expansion of module $R_0$. We first list the loop body, reset the local variable $x$, and perform then the surface with the goto statement that redirects the control flow back to location $\ell$.

The expansion of module $R_1$ is shown in Figure 8.7. Similar to module $R_0$, it does not impose any particular problems. If we would like to rename the local variables according to their surfaces, then this is now much easier since we do not have to rename anything in the depth. However, the init would still be required to disable delayed actions on the local variables, and moreover to select the right reincarnation for memorized local variables if a reaction-to-absence is required after the entering of the scope. This is not the case with the modules we considered so far, but this problem occurs in the following module:

<table>
<thead>
<tr>
<th>module $R_0$(event &amp;xOn,&amp;xOff)</th>
<th>module Expand$R_0$(event &amp;xOn,&amp;xOff)</th>
</tr>
</thead>
</table>
| loop {
  event x;
  if(x) emit xOn;
  else emit xOff;
  $\ell$:pause;
  emit x;
  if(x) emit xOn;
  else emit xOff;
} | event x;
{
if(x) emit xOn;
else emit xOff;
$\ell$:pause;
emit x;
if(x) emit xOn;
else emit xOff;
init {x};
} |
| if(x) emit xOn;
else emit xOff;
goto $\ell$; | if(x) emit xOn;
else emit xOff; |
module R1(event &xOn,&xOff) {
    loop {
        event x;
        if(x) {
            emit xOn;
            ℓ1:pause;
        } else {
            emit xOff;
            ℓ2:pause;
        }
        emit x;
        if(x) emit xOn;
        else emit xOff;
    }
}

module ExpandR1(event &xOn,&xOff) {
    event x;
    {
        if(x) {
            emit xOn;
            ℓ1:pause;
        } else {
            emit xOff;
            ℓ2:pause;
        }
    }
    emit x;
    if(x) emit xOn;
    else emit xOff;
    init {x};
}

Fig. 8.7. Curing Schizophrenia Example R1.

module R3(event a,b,int &y) {
    loop {
        int x;
        if(a) x=1; else x=2;
        ℓ1:pause;
        y=x;
        ℓ2:pause;
        if(b) x=3; else x=4;
    }
}

module ExpR3(event a,b,int &y) {
    int x;
    {
        if(a) x=1; else x=2;
        ℓ1:pause;
        y=x;
        ℓ2:pause;
        if(b) x=3; else x=4;
        init {x};
    }
    if(a) x=1; else x=2;
    goto ℓ1;
}

At starting time, the local variable has either the value 1 or 2, depending on whether the input a holds or not. The control flow enters location ℓ1 at starting time. When the control flow resumes at this location, we have to determine the value of x to transfer it to y. Clearly, the value is ether 1 or 2, depending on the previous value of a. When the control flow is then resumed
from location $\ell_2$, the local declaration of $x$ is re-entered, so that the local variable $x$ is reincarnated.

The surface/depth expansion using the goto statement is shown on the right hand side above. If we would now like to rename the local variable $x$ in the generated surface by a new name, say $x_1$, then we would have to select one of the values of $x$ or $x_1$ after we moved from location $\ell_2$ to $\ell_1$: When the control flow is at location $\ell_1$ there is no assignment to $x$, so that the reaction-to-absence requires that $x$ has to keep its previous value. This is the value that has been assigned in the most recently executed surface which is the one that assigned either 1 or 2.

Hence, the renaming would require to include also selection statements in these cases, and the construction of these statements would in turn require the complete knowledge of all guarded actions of the local variables. All in all, this would make the overall expansion more complicated, and less readable!

Next, consider the expansion of module $R_2$ as shown in Figure 8.8. Recall that the original module made two emissions of $b$ when the control flow is at location $\ell_1$ and the input $k_1$ is present. The surface/depth expansion of the previous section generated a statement that is equivalent, but that emits $b$ three times. The expansion of this section generates the module shown on the right hand side of Figure 8.8 that emits $b$ two times as the original module. Hence, the expansion of this section is closer to the original program.

Finally, consider the multiple reincarnation example of Figure 8.4. The expanded version is shown in Figure 8.9.
module ExpandMultipleReincar(event a,b,&y00,&y01,&y11, event int z) {
    event x1,x2;
    {
        weak abort
        {
            weak abort
            {
                switch
                case(!x1&!x2) do {emit y00; ℓ0:pause; z=0;}
                case(x1& x2) do {emit y01; ℓ1:pause; z=1;}
                case( x1&!x2) do {emit y10; ℓ2:pause; z=2;}
                case( x1& x2) do {emit y11; ℓ3:pause; z=3;}
                else nothing;
                emit x1;
                emit x2;
            }
            switch
            case(!x1&!x2) do {emit y00; goto ℓ0;}
            case(!x1& x2) do {emit y01; goto ℓ1;}
            case( x1&!x2) do {emit y10; goto ℓ2;}
            case( x1& x2) do {emit y11; goto ℓ3;}
            else nothing;
            when(b)
            init {x2};
        }
        switch
        case(!x1&!x2) do {emit y00; goto ℓ0;}
        case(!x1& x2) do {emit y01; goto ℓ1;}
        case( x1&!x2) do {emit y10; goto ℓ2;}
        case( x1& x2) do {emit y11; goto ℓ3;}
        else nothing;
        when(a)
        init {x1,x2};
    }
    switch
    case(!x1&!x2) do {emit y00; goto ℓ0;}
    case(!x1& x2) do {emit y01; goto ℓ1;}
    case( x1&!x2) do {emit y10; goto ℓ2;}
    case( x1& x2) do {emit y11; goto ℓ3;}
    else nothing;
}

Fig. 8.9. Surface/Depth Transformation of MultipleReincar Using goto
8.2 Formal Verification of Properties

8.2.1 From Assumptions and Assertions to Specifications

The assume and assert statements can be used to generate local conditions that finally lead to a global specification that is to be checked in the verification phase. The compiler has the task to collect these statements together with their guards so that a global specification can be generated.

Assume that we have computed the assumptions \((\beta_1, \text{assume}(\varphi_1)), \ldots, (\beta_p, \text{assume}(\varphi_p))\) and the assertions \((\gamma_1, \text{assert}(\psi_1)), \ldots, (\gamma_q, \text{assert}(\psi_q))\) for a given program. Using these assumptions and assertions, the compiler generates the following specification:

\[ \Phi_{\text{assert}} \equiv AG \left[ \left( \bigwedge_{i=1}^{p} \beta_i \rightarrow \varphi_i \right) \rightarrow \left( \bigwedge_{i=1}^{q} \gamma_i \rightarrow \psi_i \right) \right] \]

8.2.2 Model Checking of Temporal Logics

8.2.3 Supervisory Control (Reactive Synthesis)

8.3 Checking Absence of Overflows

For any bitvector assignment \(x = \tau\), we add an assertion \(\sigma\) that states that the bits that have been omitted are redundant. For array index expressions \(\tau\) in an array assignment \(x[\tau] = \pi\), we add an assertion that states that \(\tau\) is in the allowed range.

8.4 Checking Absence of Write Conflicts

- some references to related work are necessary [244, 265]

Concurrent programs may suffer from write conflicts that may occur if different values are assigned to the same variable at the same point of time. Therefore, compilers have to check that the compiled programs do not suffer from such inconsistencies. In this section, we discuss methods to check such inconsistencies for a given Quartz program and explain also why other methods that seem to be useful at a first glance can not at all be used. In particular, we show that write conflicts have essentially no correspondence to deadend states in the corresponding transition system which is not a trivial observation.

A write conflict is thereby the assignment of different values to the same variable at the same point of time. Clearly, if the variable \(x\) is assigned two different values \(\tau\) and \(\pi\), we can not satisfy both equations \(x = \tau\) and \(x = \pi\), and therefore the program runs into an inconsistent situation. The presence of
multiple threads in synchronous programs that can have complex interactions makes it difficult to see whether a program has such write conflicts. Hence, efficient algorithms to analyze a given program are required that are able to automatically check the consistency of these programs.

Since the execution of \( x = \tau; x = \pi \) with expressions \( \tau \) and \( \pi \) that evaluate to different values leads to an inconsistent situation, one might speculate that write conflicts lead to deadend states in the corresponding transition system. However, as we will demonstrate below, there is no simple relationship between the existence of deadend states in the Kripke structure and the presence of write conflicts in the program. We will demonstrate this relationship in the following with a few examples.

To this end, we use the following definitions: in general, we speak of a write conflict if two different values are assigned to a variable at the same point of time\(^2\). Since languages like Quartz and Esterel have immediate assignments and delayed assignments, we have three combinations:

- immediate/immediate write conflicts are those where two immediate assignments \( y := \tau \) and \( y := \pi \) with \( \tau \neq \pi \) are executed at the same time \( t \)
- immediate/delayed write conflicts are those where an immediate assignment \( y := \tau \) is executed at time \( t + 1 \) and a delayed assignment \( \text{next}(y) := \pi \) with \( \tau \neq \pi \) is executed at time \( t \)
- delayed/delayed write conflicts are those where two delayed assignments \( \text{next}(y) := \tau \) and \( \text{next}(y) := \pi \) with \( \tau \neq \pi \) are executed at the same time \( t \) (the write conflict occurs then at time \( t + 1 \))

For immediate/immediate write conflicts, we may furthermore distinguish if they occur at the initial point of time or after it. We will see that the effect is slightly different.

### 8.4.1 Deadend States and Write Conflicts

**Immediate/Immediate Write Conflicts at Initial Time**

In this section, we first consider immediate/immediate write conflicts that occur at initial time. The example given in Figure 8.10 demonstrates that such programs typically have corresponding Kripke structures without deadend states. As can be seen, the program has a write conflict in case that input \( x \) is false at initial time. In this case, the else-branch of the conditional statement is executed, which leads to the write conflict on the variable \( y \).

Nevertheless, the program yields a Kripke structure without deadend states: Our compiler generates the following initial condition \( I \) and the transition relation \( R \) for this program (\( \ell_0 \) is thereby the boot location [43]):

\(^2\) Note that the point of time where the write conflict occurs is therefore not the point of time where the assignment is inserted to a schedule, but the time when it is executed.
module IIWCatInit:
    input x;
    output y:boolean;
    if x then ell_1:halt
    else
        y := true;
        y := false
    end
end

Fig. 8.10. Immediate/Immediate Write Conflicts at Initial Time hide Deadend States

\[
\begin{align*}
    I & \equiv \left( (\ell_0 \land \neg x \rightarrow y) \land (\ell_0 \land \neg x \rightarrow \neg y) \land (\neg (\ell_0 \land \neg x) \rightarrow \neg y) \land (\ell_0 \leftrightarrow 1) \land (\ell_1 \leftrightarrow 0) \right) \\
    R & \equiv \left( (\ell_0 \land \neg x \rightarrow y) \land (\ell_0 \land \neg x \rightarrow \neg y) \land (\neg (\ell_0 \land \neg x) \rightarrow (\operatorname{next}(\ell_0) \leftrightarrow 0) \land (\operatorname{next}(\ell_1) \leftrightarrow \ell_1 \lor \ell_0 \land x) \right)
\end{align*}
\]

It is easily seen that $I$ is equivalent to $\ell_0 \land \neg \ell_1 \land x \land \neg y$, so that we only have one initial state. The transition relation $R$ encodes the six transitions given in Figure 8.10. Hence, the obtained Kripke structure does not have deadend states.

To discuss this example in more detail, recall that states of Kripke structures describe entire situations that can occur during the computations of the program. However, this directly implies that only consistent situations can appear in the Kripke structure, since every state of the Kripke structure is labeled with a (consistent) variable assignment. As a consequence, situations where immediate/immediate write conflicts occur are not represented in the corresponding Kripke structure. If this situation occurs at initial time, and no further write conflicts occur, then the Kripke structure has no deadend states.

As program IIWCatInit has a write conflict in case input $x$ would be false at initial time, there is no such initial state in the Kripke structure. It would be impossible to define a truth value for $y$ to satisfy the initial condition $I$. For this reason, the problematic situation does not occur in the Kripke structure.

**Immediate/Immediate Write Conflicts after Initial Time**

In this section, we consider immediate/immediate write conflicts that occur after initial time. The example given in Figure 8.11 demonstrates that such programs typically have corresponding Kripke structures with deadend states. As can be seen, the program has a write conflict in case that input $x$ is false when the control is at location ell_1.

If the write conflict does not already occur at initial time, then there is a consistent state that leads to the write conflict. In this case, the Kripke structure will have deadend states. An example is shown in Figure 8.11. Our com-
8.4 Checking Absence of Write Conflicts

module IIWafterInit:
  input x;
  output y:boolean;
  ell_1:pause;
  if x then
    y := true;
    y := false
  end
end

Fig. 8.11. Immediate/Immediate Write Conflicts after Initial Time yield Deadend States

piler generates the following initial condition \( I \) and the transition relation \( R \) for this program (\( \ell_0 \) is again the boot location):

\[
I := \left\{ \begin{array}{l}
(\ell_1 \land x \rightarrow y) \\
(\ell_1 \land x \rightarrow \neg y) \\
(\neg(\ell_1 \land x) \rightarrow \neg y) \\
(\ell_0 \leftrightarrow 1) \\
(\ell_1 \leftrightarrow 0)
\end{array} \right.
\]

\[
R := \left\{ \begin{array}{l}
(\ell_1 \land x \rightarrow y) \\
(\ell_1 \land x \rightarrow \neg y) \\
(\neg\text{next}((\ell_1 \land x)) \rightarrow (\text{next}(y) \leftrightarrow y)) \\
(\text{next}(\ell_0) \leftrightarrow 0) \\
(\text{next}(\ell_1) \leftrightarrow \ell_0)
\end{array} \right.
\]

It is easily seen that \( I \) is equivalent to \( \ell_0 \land \neg \ell_1 \land \neg y \), so that we have two initial states. The transition relation \( R \) encodes the transitions given in Figure 8.11. To see how the deadend states are generated, let us first instantiate the initial condition in the transition relation:

\[
R^0 := \left\{ \begin{array}{l}
(\neg\text{next}(x) \rightarrow (\text{next}(y) \leftrightarrow y)) \\
(\text{next}(\ell_0) \leftrightarrow 0) \\
(\text{next}(\ell_1) \leftrightarrow 1)
\end{array} \right.
\]

This explains why there is no transition from the initial point of time to a state where \( \neg \ell_0 \land \ell_1 \land \neg x \land y \) holds (since the first conjunct of \( R^0 \) forbids this). Now, let us instantiate \( \ell_0 = 0 \) and \( \ell_1 = 1 \) in \( R \) to obtain a symbolic description of the transitions that are possible at time \( t = 1 \):

\[
R^1 := \left\{ \begin{array}{l}
(x \rightarrow y) \\
(x \rightarrow \neg y) \\
(\text{next}(y) \leftrightarrow y) \\
(\text{next}(\ell_0) \leftrightarrow 0) \\
(\text{next}(\ell_1) \leftrightarrow 0)
\end{array} \right.
\]

How can we explain the situation? Clearly, if the control flow is at location \( \ell_1 \) then \( x \) must not hold, since otherwise we obtain the write conflict that is an inconsistent situation. However, our transition relation allows us to reach such a state, but it does not provide transitions to leave such a state. We could
fix that behaviour by modifying the transition relation in the general case as follows, so that states where an immediate/immediate write conflict occurs cannot be reached:

\[
\mathcal{R}(x, \text{next}(x)) := \left\{ \begin{array}{c}
\left( \bigwedge_{j=1}^{p} (\gamma_j \to x = \tau_j) \right) \land \text{next} \left( \bigwedge_{j=1}^{p} (\gamma_j \to x = \tau_j) \right) \\
\left( \bigwedge_{j=1}^{q} (\chi_j \to \text{next}(x) = \pi_j) \right) \land \\
\text{next} \left( \bigwedge_{j=1}^{p} \neg \gamma_j \right) \land \\
\left( \bigwedge_{j=1}^{q} \neg \chi_j \right) \to \text{next}(x) = x
\end{array} \right. 
\]

However, as we are only interested in the infinite paths\(^3\) of the Kripke structure, there is no reason to change the transition relation. Moreover, not adding the additional formula allows us to detect such write conflicts by checking for deadend states!

### Delayed/Delayed Write Conflicts

In this section, we consider delayed/delayed write conflicts. For these actions, it does not matter whether they are executed at or after initial time, since the caused write conflict will never occur at initial time. The example given in Figure 8.12 demonstrates that such programs typically have corresponding Kripke structures with deadend states. As can be seen, the program has a write conflict in case that input \(x\) is true at initial time.

```plaintext
module DDWC:
  input x;
  output y:boolean;
  if x then
    next(y) := true;
    next(y) := false
  end
end
```

**Fig. 8.12.** Delayed/Delayed Write Conflicts yield Kripke Structures with Deadend States

\(^3\) Note that also computations of programs that terminate after some time will lead to infinite paths in the Kripke structure since final states where all control flow locations are inactive have self-loops and loops to other states whose labels differ only on input variables.
Our compiler generates the following initial condition \( I \) and the transition relation \( R \) for this program (\( \ell_0 \) is again the boot location):

\[
I := (\neg y \land \ell_0)
R := \begin{cases}
(\ell_0 \land x \rightarrow \text{next}(y)) \land \\
(\ell_0 \land x \rightarrow \neg\text{next}(y)) \land \\
(\neg\text{next}((\ell_0 \land x)) \rightarrow (\text{next}(y) \leftrightarrow y)) \land \\
(\text{next}(\ell_0) \leftrightarrow 0)
\end{cases}
\]

It is easily seen that \( I \) is equivalent to \( \ell_0 \land \neg \ell_1 \land \neg y \), so that we have two initial states that differ on the value of the input \( x \). The transition relation \( R \) encodes the transitions given in Figure 8.12. If we instantiate the initial condition in the transition relation, we obtain the following transitions:

\[
R^0 := \begin{cases}
(x \rightarrow \text{next}(y)) \land \\
(x \rightarrow \neg\text{next}(y)) \land \\
(\neg\text{next}(x) \rightarrow (\text{next}(y) \leftrightarrow y)) \land \\
(\text{next}(\ell_0) \leftrightarrow 0)
\end{cases}
\]

Hence, only one of the initial states has transitions, since the other one schedules assignments that would lead to a write conflict after leaving the state. It is easily seen that this is typical for D/D write conflicts, so that we can detect them by checking for deadend states.

**Immediate/Delayed Write Conflicts**

In this section, we consider immediate/delayed write conflicts. Similar to delayed/delayed write conflicts, it does not matter whether they are executed at or after initial time, since the caused write conflict will never occur at initial time. The example given in Figure 8.13 demonstrates that such programs may have deadend states in their corresponding Kripke structure.

```plaintext
module IDWC:
    input x;
    output y:boolean;
    next(y) := true;
    ell_1:pause;
    if x then
        y := false
    end
end
```

**Fig. 8.13.** Programs without Write Conflicts may yield Kripke Structures with Deadend States

Our compiler generates the following initial condition \( I \) and the transition relation \( R \) for this program (\( \ell_0 \) is again the boot location):
It is easily seen that $I$ is equivalent to $\ell_0 \land \neg \ell_1 \land \neg y$, so that we have two initial states that differ on the value of the input $x$. The transition relation $R$ encodes the transitions given in Figure 8.13. If we instantiate $\ell_0 = 0$ and $\ell_1 = 1$ in the transition relation, we obtain the following transitions:

$$R^1 := \begin{pmatrix}
(x \rightarrow \neg y) \\
(next(y) \leftrightarrow y) \\
(next(\ell_0) \leftrightarrow 0) \\
(next(\ell_1) \leftrightarrow 0)
\end{pmatrix}$$

Hence, we have a similar situation as with immediate/immediate write conflicts that happen after initial time: we can reach states to do not behave consistently, but it is not possible to leave such states. As already discussed, we could modify the transition relation to fix that problem. However, this is not necessary, in terms of the correctness of the Kripke structure: In any case, all infinite paths of the Kripke structure correspond with correct computations of the program. Deadend paths simply have no meaning.

**Deadend States without Write Conflicts**

In the previous sections, we discussed if and how different kinds of write conflicts lead to deadend states in the corresponding Kripke structure. In this section, we consider the converse implication: is it possible that a Kripke structure has deadend states, even if the program is free of write conflicts? The surprising result is that this is really the case, as we will demonstrate with the example given in Figure 8.14.

```
module NOWC:
  input x;
  output y:boolean;
  loop
    y := x;
    ell_1:pause
  end
end
```

Fig. 8.14. Programs without Write Conflicts may yield Kripke Structures with Deadend States
Our compiler generates the following initial condition $\mathcal{I}$ and the transition relation $\mathcal{R}$ for this program ($\ell_0$ is again the boot location):

$$\mathcal{I} := \begin{cases} (\ell_0 \rightarrow (y \leftrightarrow x)) \land \\ (\ell_1 \rightarrow (y \leftrightarrow x)) \land \\ (¬(\ell_0 \lor \ell_1) \rightarrow ¬y) \land \\ (\ell_0 \leftrightarrow 1) \land \\ (\ell_1 \leftrightarrow 0) \end{cases} \quad \mathcal{R} := \begin{cases} (\ell_0 \rightarrow (y \leftrightarrow x)) \land \\ (\ell_1 \rightarrow (y \leftrightarrow x)) \land \\ (¬next(\ell_0 \lor \ell_1) \rightarrow (next(y) \leftrightarrow y)) \land \\ (next(\ell_0) \leftrightarrow 0) \land \\ (next(\ell_1) \leftrightarrow \ell_0 \lor \ell_1) \end{cases}$$

It is easily seen that $\mathcal{I}$ is equivalent to $\ell_0 \land ¬\ell_1 \land (y \leftrightarrow x)$, so that we have two initial states that differ on the value of the input $x$. The transition relation $\mathcal{R}$ encodes the transitions given in Figure 8.14. If we instantiate the initial condition in the transition relation, we obtain the following transitions (which even hold for all points of time):

$$\mathcal{R}^0 := \begin{cases} (y \leftrightarrow x) \land \\ (next(\ell_0) \leftrightarrow 0) \land \\ (next(\ell_1) \leftrightarrow 1) \end{cases}$$

Hence, we have a similar situation as with immediate/immediate write conflicts that happen after initial time: we can reach states to do not behave correctly or consistently, but it is not possible to leave such states. As already discussed, we could modify the transition relation to fix that problem. However, this is not necessary, in terms of the correctness of the Kripke structure: in any case, all infinite paths of the Kripke structure correspond with correct computations of the program. Deadend paths simply have no meaning.

The above problem does also arise when Mealy automata have to be translated to Kripke structures before model checking procedures can be applied for verification. An example is shown in Figure 8.15.

**Fig. 8.15.** Construction of Kripke Structures for Mealy Automata

The example given in Figure 8.15 shows a simple hardware circuit without write conflicts: its behavior is easily described with the Mealy automaton given below it. However, if we use the conjunction of its transition equation and output equation as transition relation of a Kripke structure, then we obtain the Kripke structure with deadend states given in the Figure. Clearly, the deadend states describe situations that never appear in the Mealy automaton. Nevertheless, they appear in the Kripke structure.
A common way\textsuperscript{4} to circumvent the problem for Mealy automata is to define the Kripke structure first without outputs of the Mealy automaton. This yields the Kripke structure given in the lower row of Figure 8.15, which has no deadend states. After this, we label for each output equation $y_i := \varphi_i$, those states with $y_i$ that satisfy $\varphi_i$.

In our compiler, we use this strategy when equational code is generated. In this case, it is easily seen that no deadend states can occur in the Kripke structure (causality problems must be excluded for this kind of code generation, since the equations must be ordered in terms of their dependencies).

### 8.4.2 Detection of Write Conflicts

The examples given in the previous sections show that there is no direct relationship between deadend states and write conflicts. In particular, we have found the following results:

<table>
<thead>
<tr>
<th>example</th>
<th>deadends</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIWCatInit</td>
<td>no</td>
</tr>
<tr>
<td>IIWCafterInit</td>
<td>yes</td>
</tr>
<tr>
<td>DDWC</td>
<td>yes</td>
</tr>
<tr>
<td>IDWC</td>
<td>yes</td>
</tr>
<tr>
<td>NOWC</td>
<td>yes</td>
</tr>
</tbody>
</table>

As it is possible to describe the computations of a program with different Kripke structures (since only the infinite paths are of interest), there are different ways to derive a transition relation. We already discussed a modification that additionally takes the immediate actions of the next point of time into account. Using this transition relation, we would obtain the following results:

<table>
<thead>
<tr>
<th>example</th>
<th>deadends</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIWCatInit</td>
<td>no</td>
</tr>
<tr>
<td>IIWCafterInit</td>
<td>no</td>
</tr>
<tr>
<td>DDWC</td>
<td>yes</td>
</tr>
<tr>
<td>IDWC</td>
<td>no</td>
</tr>
<tr>
<td>NOWC</td>
<td>no</td>
</tr>
</tbody>
</table>

Hence, we could use deadend computation to check for potential delayed/delayed write conflicts. However, also other kinds of write conflicts can yield deadend states, but as the above table shows, they need not necessarily do so. Using the modified transition relation, we could prove the following: \textit{if an obtained}

\textsuperscript{4} This procedure can, however, not be applied to synchronous programs. The problem is that these programs may yield cyclic equations, i.e., it may be the case that $y_i$ occurs in $\varphi_i$. For this reason, we are forced to consider the Kripke structure with deadend states, and have to find a criterion for this structure that tells us whether the modeled system has write conflicts or not.
8.4 Checking Absence of Write Conflicts

Kripke structure has deadend states, then it has write conflicts. However, the converse would still not hold, as examples IIWCatInit and IIWCAfterInit demonstrate.

So, the next question is how then, can write conflicts be detected? There are several possible ways to check write conflicts:

- we generate equational code, where potential write conflicts have already been resolved by giving the actions priorities by using case-statements and check with that description that there is no variable where two or more guards of assignments are active at the same point of time
- we use quantified \(\mu\)-calculus to check that the formula \(\forall \overset{\rightarrow}{i}.\Diamond 1\) holds in every reachable state, i.e., that there is a successor state for each input.

To implement the first solution, we have to define what a write conflict is. Informally, a write conflict appears if two or more different values are assigned to a variable at the same time. According to the computed guarded commands, this leads to the following temporal (path) formula [223]:

\[
\text{WriteConflict}(x) \equiv \neg \left( \bigwedge_{i=1}^{p} \bigwedge_{j=i+1}^{p} \gamma_i \land \gamma_j \rightarrow \tau_i = \tau_j \right) \land \\
\left( \bigwedge_{i=1}^{q} \bigwedge_{j=i+1}^{q} \chi_i \land \chi_j \rightarrow \pi_i = \pi_j \right) \land \\
\left( \bigwedge_{i=1}^{p} \bigwedge_{j=1}^{p} \chi_i \land \text{next}(\gamma_j) \rightarrow \pi_i = \text{next}(\tau_j) \right)
\]

It is simple to generate this formula as a specification for a model-checker that can be directly checked after compilation for every (output/local) state variable. Note that the formula is a path formula, however, it can be checked in polynomial time.

The other solution is to use a specification in the quantified \(\mu\)-calculus. This is a more powerful solution, since it can also be used to check causality conflicts.

Fig. 8.16. Programs without Write Conflicts may yield Kripke Structures with Deadend States

Program NOWC yields the following equations with the definition that \(y \equiv (\ell_0 \rightarrow x) \land (\neg \ell_0 \land \ell_1 \rightarrow x) \land (\neg \ell_0 \land \neg \ell_1 \rightarrow y')\)
module NOWC:
input x;
output y:boolean;
loop
y := x;
nell_1:pause
end
end

init(ℓ₀) = 1  next(ℓ₀) = 0
init(ℓ₁) = 0  next(ℓ₁) = ℓ₀ ∨ ℓ₁
init(y) = x  next(y') = (ℓ₀ → x)∧
                     (¬ℓ₀ ∧ ℓ₁ → x)∧
                     (¬ℓ₀ ∧ ¬ℓ₁ → y')

The construction of the Kripke structure can be easily seen by the following specializations of the transition relation:

<table>
<thead>
<tr>
<th>ℓ₀ℓ₁</th>
<th>ℓ₀ℓ₁</th>
<th>ℓ₀ℓ₁</th>
<th>ℓ₀ℓ₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>next(ℓ₀) = 0</td>
<td>next(ℓ₀) = 0</td>
<td>next(ℓ₀) = 0</td>
<td>next(ℓ₀) = 0</td>
</tr>
<tr>
<td>next(ℓ₁) = 0</td>
<td>next(ℓ₁) = 1</td>
<td>next(ℓ₁) = 1</td>
<td>next(ℓ₁) = 1</td>
</tr>
<tr>
<td>next(y') = y'</td>
<td>next(y') = x</td>
<td>next(y') = x</td>
<td>next(y') = x</td>
</tr>
</tbody>
</table>

Figure 8.16 shows the Kripke structure that is encoded with the equation system. Reachable states are given in green color and states that violate the invariant are given in red color. White states are therefore unreachable states that satisfy however the invariant. These states appear as overapproximation of the reachable states by the invariant.

The transition relation can therefore be modified in that transitions that start in red states can be eliminated or added, while all other transitions have to be retained.

The specification to be checked is AG¬(ℓ₀ ∧ ℓ₁), or equivalently ¬EF(ℓ₀ ∧ ℓ₁), which is translated to the µ-calculus formula ¬µy. (ℓ₀ ∧ ℓ₁ ∧ Φinf) ∨ ◊y. As there are no deadend states, it follows that Φinf corresponds with the set of all states, and therefore our fixpoint reduces to ¬µy. (ℓ₀ ∧ ℓ₁) ∨ ◊y, which means non-reachability of the red states.

Question: Is there also a minimization of the initial states? This would be wrong since it would allow the BDD package to add arbitrarily many of the red states to the initial state set! On the other hand, using the restrict operation for minimization of the state sets that are obtained during model checking is okay.

Interestingly, Cadence-SMV only counts four reachable states, since it only considers ℓ₀ and ℓ₁ as state holding variables, while x and y are ignored. The reason for this is that the specification ¬EF(ℓ₀ ∧ ℓ₁) requires only to consider at the subsystem that is defined via the equations of ℓ₀ and ℓ₁, and since these variables do not depend on x or y, we can ignore these variables.
9

Hardware-Software Codesign

9.1 Introduction

9.2 Application-Specific Microprocessors

9.3 Dynamically Reconfigurable Processing Elements

9.4 HW/SW-Codesign based on Virtual Platforms
A Partial Orders

The formal foundation of the semantics of most models of computation are based on the existence and computation of certain fixpoints. In this appendix, we therefore provide the necessary mathematical background required for the definition of these semantics. In particular, we study in Section A.1 complete partial orders as the general framework, and in Section A.2 its specialization to complete lattices. Applications of these theories are found in Appendix 7 to define the semantics of dataflow process networks, and in Section 4 to define the constructive semantics of the Quartz language, as well as in Section 4.6 on causality analysis. This appendix only covers some special theorems needed for these applications. More details on the mathematics of partial orders can be found in related textbooks and survey articles like [1, 73, 219].

A.1 Complete Partial Orders

In this section, we recall basic definitions and facts about partial orders, least and greatest upper bounds, as well as monotone and continuous functions. These definitions are shared by lattices and complete partial orders.

A.1.1 Partial Orders and Directed Subsets

**Definition A.1 (Total and Partial Orders).** For every set $D \neq \emptyset$ with a relation $\sqsubseteq$ on $D \times D$, the pair $(D, \sqsubseteq)$ is a partial order if the following laws hold:

- $\forall x \in D. \ x \sqsubseteq x$ (reflexivity)
- $\forall x, y \in D. \ x \sqsubseteq y \land y \sqsubseteq x \rightarrow x = y$ (antisymmetry)
- $\forall x, y, z \in D. \ x \sqsubseteq y \land y \sqsubseteq z \rightarrow x \sqsubseteq z$ (transitivity)

$(D, \sqsubseteq)$ is a total order if all elements can be compared, i.e., if we additionally have $\forall x, y \in D. \ x \sqsubseteq y \lor y \sqsubseteq x$. 
Total orders are often also called linear orders, which is however misleading, since it may not be the case that the elements of a total order can be arranged in a sequence: As a counterexample, consider the set $\mathbb{B} \times \mathbb{N}$ with the following relation:

$$(b_1, n_1) \sqsubseteq (b_2, n_2) :\iff \begin{cases} n_1 \leq n_2 : & \text{if } b_1 = b_2 \\
 b_1 \land \neg b_2 : & \text{if } b_1 \neq b_2 \end{cases}$$

The elements of this total order can be drawn essentially as two linear sequences where all elements of one sequence are greater than all elements of the other sequence. Since both sequences are infinite, we can not concatenate them to a single sequence. Somehow, we can view this total order as the concatenation of two infinite sequences.

**Definition A.2 (Chains).** A subset $\{\} \neq C \subseteq D$ of a partial order $(D, \sqsubseteq)$ where $(C, \sqsubseteq)$ is a total order is called a chain.

According to the above example, the notion ‘chain’ is in the same way misleading as linear orders. A chain must not necessarily be a linear sequence, as the following characterization of chains reveals:

**Theorem A.3 (Total Orders).** The partial order $(D, \sqsubseteq)$ is total iff every nonempty finite subset $M \subseteq D$ has a maximum.

**Proof.** By induction, we first prove that every finite total order has a maximum. The induction base is trivial. In the induction step, we have to determine the maximum of a total order $(M \cup \{e\}, \sqsubseteq)$, and we know by the induction hypothesis that $(M, \sqsubseteq)$ has a maximum $m \in M$. Since $(M \cup \{e\}, \sqsubseteq)$ is a total order, we either must have $m \sqsubseteq e$ or $e \sqsubseteq m$, so that either $e$ or $m$ is the maximum of $M \cup \{e\}$.

Hence, the implication from the left to the right immediately follows, since every finite subset $M \subseteq D$ of a total order is also totally ordered, and thus must have a maximum. The other direction is even simpler: if $(D, \sqsubseteq)$ is not total, then there are elements $x, y \in D$ with $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$, and therefore the set $\{x, y\}$ has no maximum. \(\square\)

Note that in the problem imposed by the above example, where two infinite sequences should have been concatenated breaks down if we restrict the consideration to finite sets only. Hence, a consequence of the above theorem is that each finite totally ordered set can be arranged in a sequence.

**Definition A.4 (Least Upper and Greatest Lower Bounds).** Let $(D, \sqsubseteq)$ be a partial order. For any set $M \subseteq D$, an element $m_u \in D$ is an upper bound, if $\forall x \in M. x \sqsubseteq m_u$ holds. Moreover, $m_u$ is the least upper bound of $M$ if the following holds:

$$(\forall x \in M. x \sqsubseteq m_u) \land (\forall y \in D.(\forall x \in M. x \sqsubseteq y) \rightarrow m_u \sqsubseteq y)$$

Similarly, $m_l \in D$ is a lower bound of $M$, if $\forall x \in M. m_l \sqsubseteq x$ holds, and it is the greatest lower bound of $M$ if the following holds:
\[(\forall x \in M. m_1 \sqsubseteq x) \land (\forall y \in D. (\forall x \in M. y \sqsubseteq x) \rightarrow y \sqsubseteq m_1)\]

We denote the least upper and the greatest lower bound of \(M\) as \(\sup(M)\) and \(\inf(M)\), respectively, and often use the alternative names ‘supremum’ and ‘infimum’. \(\sup(M)\) is moreover the maximal element of \(M\), denoted as \(\max(M)\), if \(\sup(M) \in M\) holds. Analogously, \(\inf(M)\) is the minimal element of \(M\), denoted as \(\min(M)\), if \(\inf(M) \in M\) holds.

Of course, a subset \(M \subseteq D\) needs neither have lower nor upper bounds, and even if such bounds exist, there is no need for the existence of the supremum or the infimum. For example, the sets of rational and real numbers are totally ordered with the usual ordering relation. The set \(M := \{x \in \mathbb{Q} \mid \sqrt{2} \leq x < \sqrt{3}\}\) has upper and lower bounds, but neither a supremum nor an infimum in \(\mathbb{Q}\). However, if either \(\sup(M)\) or \(\inf(M)\) exists, then it is uniquely determined (by the reflectivity of \(\sqsubseteq\)).

**Lemma A.5 (Simple Properties of Bounds).** For every partially ordered set \((D, \sqsubseteq)\) and subsets \(A, B \subseteq D\), we have:

- \(\sup(A) = \sup(B)\) iff \(\sup(A)\) is an upper bound of \(B\) and \(\sup(B)\) is an upper bound of \(A\).
- \(\inf(A) = \inf(B)\) iff \(\inf(A)\) is a lower bound of \(B\) and \(\inf(B)\) is a lower bound of \(A\).
- \(\sup(\bigcup_{i \in I} A_i) = \sup(\bigcup_{i \in I} \sup(A_i))\), where \(I\) is an arbitrary index set.
- \(\inf(\bigcap_{i \in I} A_i) = \inf(\bigcap_{i \in I} \inf(A_i))\), where \(I\) is an arbitrary index set.
- \(A \subseteq B\) implies \(\sup(A) \sqsubseteq \sup(B)\).
- \(A \subseteq B\) implies \(\inf(B) \sqsubseteq \inf(A)\).

**Proof.** Consider the first point: The direction from the left to the right is trivial. The other direction is seen as follows: If \(\sup(A)\) is an upper bound of \(B\), it immediately follows that \(\sup(B) \sqsubseteq \sup(A)\) holds, since \(\sup(B)\) is the least upper bound of \(B\). Analogously, we conclude that \(\sup(A) \sqsubseteq \sup(B)\) holds, if \(\sup(B)\) is an upper bound of \(A\). By antisymmetry of \(\sqsubseteq\), it therefore follows that \(\sup(A) = \sup(B)\). The second point is proved analogously, and the other points are proved by the help of the first two. \(\Box\)

In general, we do not have a reasonable relationship between least upper and greatest lower bounds. In particular, the equation \(\sup(M) = \inf(D \setminus M)\) does not hold in general: For example, in the totally ordered set \((\mathbb{R}, \leq)\), the set \(M := \{x \in \mathbb{Q} \mid \sqrt{2} \leq x < \sqrt{3}\}\) has \(\inf(M) = \sqrt{2}\) and \(\sup(M) = \sqrt{3}\), but the complement set \(\mathbb{R} \setminus M\) neither has a lower nor an upper bound.

In analysis, one considers the convergence of sequences to certain limits. Mathematical order theory generalizes this in that sequences are replaced by more general directed subsets that are defined as follows:

**Definition A.6 (Directed Subsets).** Let \((D, \sqsubseteq)\) be a partially ordered set. A nonempty subset \(\{\}\ \neq M \subseteq D\) is a directed subset if all two-element sets \(\{x, y\} \subseteq M\) have an upper bound in \(M\).
Clearly, all subsets of a totally ordered set are directed, since for arbitrary
two-element sets \( \{x, y\} \), one of the elements is the maximum and the other
one is the minimum of the two-element set. Directed sets are in some sense a
generalization of sequences that can be used to reason about convergence: in
a directed set \( M \), we can find for two elements \( x_0 \) and \( x_1 \) an upper bound \( x_2 \)
in \( M \), and then an upper bound \( x_3 \) for \( x_1 \) and \( x_2 \), and so on. Note further that
a directed subset may not have a supremum, which however exists even as a
maximum in case the directed subset is finite:

**Lemma A.7 (Finite Directed Subsets).** Let \( (D, \sqsubseteq) \) be a partially ordered set,
and \( M \subseteq D \) be a nonempty finite subset of \( D \). Then, \( M \) is a directed subset iff it
has a maximal element.

**Proof.** Clearly, if \( M \) has a maximal element, then \( M \) is a directed subset, since
that maximal element is an upper bound for each pair \( \{x, y\} \subseteq M \). The converse implication is proved by contradiction: Assume \( \{\} \subseteq X \subseteq D \) would be
finite and directed subset, hence, we have (1) \( \forall x_1, x_2 \in X. \exists x_3 \in X. \ x_1 \preceq \ x_3 \land \ x_2 \preceq \ x_3 \). Assume further that \( X \) has no maximum. Hence, we have (2)
\( \forall x_1 \in X. \exists x_2 \in X. \neg(x_2 \preceq x_1) \). Now, consider an element \( x_0 \in X \). By (2), we
conclude that there is a \( y_0 \in X \) with \( \neg(x_0 \preceq y_0) \). By (1), we know that there is a \( x_1 \in X \) with \( x_0 \preceq x_1 \land y_0 \preceq x_1 \), and we have \( x_1 \neq x_0 \), since otherwise we
would have a contradiction to \( \neg(x_0 \preceq y_0) \). Again, by (2), we conclude that there is a \( y_1 \in X \) with \( \neg(x_1 \preceq y_1) \), and by (1), there is a \( x_2 \in X \) with \( x_1 \preceq x_2 \land y_1 \preceq x_2 \), and we have \( x_2 \neq x_1 \), since otherwise we would have
a contradiction to \( \neg(x_1 \preceq y_1) \). By repeating these arguments, we construct a
strictly increasing sequence of values \( x_i \in X \) which is not possible due to the
finiteness of \( X \).

Hence, finite directed subsets correspond to (reflexive-transitive closures of)
directed acyclic graphs with a single root node. Note that a consequence of
this lemma is that not every subset of a directed subset is also a directed
subset. However, if \( M \) is a directed subset, then its downward closure is also
a directed subset:

**Lemma A.8 (Directed Subsets and Downward Closures).** Let \( (D, \sqsubseteq) \) be a partially ordered set and \( M \) a nonempty subset of \( D \). Define the downward closure
down(\( M \)) of \( M \) as \( \text{down}(M) := \{x \in D \mid \exists y \in M.x \preceq y\} \), i.e., the set of
elements less than or equal to elements of \( M \). Then, \( M \) is a directed subset iff
down(\( M \)) is a directed subset of \( D \).

**Proof.** First note that \( M \subseteq \text{down}(M) \) holds by definition of \( \text{down}(M) \) and
reflexivity of \( \sqsubseteq \).

Assume first that \( M \) is a directed subset, and assume that \( x, y \in \text{down}(M) \).
By the definition of \( \text{down}(M) \), there exist \( \hat{x}, \hat{y} \in M \) with \( x \preceq \hat{x} \) and \( y \preceq \hat{y} \),
and by the direction of \( M \) there is a \( z \in M \) with \( \hat{x} \preceq z \) and \( \hat{y} \preceq z \). Thus, by
transitivity of \( \sqsubseteq \), we therefore have \( x \preceq z \) and \( y \preceq z \), so that also \( \text{down}(M) \) is
directed (note that \( M \subseteq \text{down}(M) \) holds).
To prove the converse, assume that $\text{down}(M)$ is a directed subset, and that $x, y \in M$ holds. Since $M \subseteq \text{down}(M)$ holds, we also have $x, y \in \text{down}(M)$, and since $\text{down}(M)$ is a directed subset, there is a $\hat{z} \in \text{down}(M)$ with $x \sqsubseteq \hat{z}$ and $y \sqsubseteq \hat{z}$. Since $\hat{z} \in \text{down}(M)$ holds, there is by the definition of $\text{down}(M)$ an element $z \in M$ such that $\hat{z} \sqsubseteq z$. By transitivity of $\sqsubseteq$, we therefore have $x \sqsubseteq z$ and $y \sqsubseteq z$, so that $M$ is a directed subset. 

Hence, for arbitrary subsets $M \subseteq D$, $\text{down}(M)$ may not be a directed subset. This is the case if and only if $M$ is already a directed subset.

A.1.2 Complete Partial Orders

Based on directed subsets, we can now demand the completeness of a partial order that guarantees the existence of fixpoints of certain functions.

**Definition A.9 (Complete Partial Orders).** A partially ordered set $(D, \sqsubseteq)$ is a complete partial order if all directed subsets $M \subseteq D$ have a supremum $\text{sup}(M)$ in $D$, and if there is a minimal element $\bot \in D$.

‘Complete partial order’ is often abbreviated as cpo. Note that $\bot = \text{sup}(\emptyset)$ holds, but since $\emptyset$ is not a directed set, we have to demand the existence of $\bot = \text{sup}(\emptyset)$ explicitly. According to Lemma A.7, every finite directed subset has even a maximum. Hence, we have the following immediate consequence:

**Theorem A.10 (Finite Partial Orders).** Every finite partial order that has a minimal element is complete.

The proof follows directly from Lemma A.7, since in a finite partial order, all directed subsets must also be finite, and thus, they do even have maxima. Some textbooks and articles define complete partial orders with weaker restrictions, namely as so-called chain-complete partial orders, where only the chains in $D$ must have suprema. It is astonishing that this seemingly strong restriction does not make a difference:

**Theorem A.11 (Chain Theorem).** A partial order $(D, \sqsubseteq)$ is complete iff every chain in $D$ has a supremum.

This theorem is however remarkably difficult to prove. The known proofs including the one mentioned in [174, 219] make use of the axiom of choice or its equivalent formulations in terms of Zorn’s lemma or the Hausdorff maximal principle. We omit the proof, and refer the interested reader to [174, 219].

The most important property of functions on partial orders is certainly that such a function preserves the ordering relation of elements, i.e., that the function is monotone. There is however also a stronger requirement called continuity where not only the ordering relation between elements is maintained, but where also suprema of sets are mapped to the suprema of their images:
Definition A.12 (Monotone and Continuous Functions). Given partially ordered sets \((D, \sqsubseteq_D)\) and \((E, \sqsubseteq_E)\), and a function \(f : D \to E\). \(f\) is monotone if the following holds:

\[
\forall x, y \in D. \ x \sqsubseteq_D y \implies f(x) \sqsubseteq_E f(y).
\]

If \((D, \sqsubseteq_D)\) and \((E, \sqsubseteq_E)\) are moreover complete, then \(f\) is continuous if it maps every directed subset \(M \subseteq D\) to a directed subset \(f(M) \subseteq E\), and we moreover have \(f(\text{sup}(M)) = \text{sup}\{f(x) \mid x \in M\}\).

Some definitions of continuous functions additionally demand that continuous functions should be monotone. As we prove below, this is however not necessary, since monotonicity already follows from continuity:

Lemma A.13 (Continuity implies Monotonicity).
Every continuous function is monotone.

Proof. Assume \(f : D \to E\) is continuous and \(x \sqsubseteq y\) holds. Thus, we have \(y = \text{sup}\{x, y\}\), and thus \(f(y) = f(\text{sup}\{x, y\})\). \(\{x, y\}\) is directed, since it is totally ordered. Hence, since \(f\) is continuous, we have \(f(\text{sup}\{x, y\}) = \text{sup}\{f(x), f(y)\}\). Combining the last two equations yields \(\text{sup}\{f(x), f(y)\} = f(y)\), which implies \(f(x) \sqsubseteq f(y)\). \(\Box\)

The converse is in general not true, but it holds in all finite cpos. The inequalities \(\text{sup}\{f(x) \mid x \in M\} \sqsubseteq_E f(\text{sup}(M))\) and \(f(\text{inf}(M)) \sqsubseteq_E \text{inf}\{f(x) \mid x \in M\}\), however, already hold for all \(M \subseteq D\) and all monotone functions \(f\) (provided that the infima and suprema exist). Moreover, considering functions \(f : D \to D\), certain elements \(x \in D\) are of special interest:

- \(x\) is fixpoint of \(f : D \to D\) if \(f(x) = x\) holds
- \(x\) is pre-fixpoint of \(f : D \to D\) if \(f(x) \sqsubseteq x\) holds
- \(x\) is post-fixpoint of \(f : D \to D\) if \(x \sqsubseteq f(x)\) holds

If a function \(f\) has even a least or a greatest fixpoint, we denote the least fixpoint of \(f\) as \(\mu x. f(x)\) and the greatest fixpoint as \(\nu x. f(x)\). Clearly, every fixpoint is also a pre-fixpoint as well as a post-fixpoint, and conversely the intersection of the set of pre- and post-fixpoints is the set of fixpoints of a function \(f\). However, not every function \(f\) may have fixpoints. In a complete partial \((D, \sqsubseteq)\) order, however, every monotone function \(f : D \to D\) has fixpoints, which is stated in Theorem A.17 below. For its proof, we need to prove a lemma about inflationary functions according to [219] (shorter proofs make use of transfinite induction).

Lemma A.14 (CPO of Endomorphisms). Let \((D, \sqsubseteq)\) be a complete partial order and let \(\mathcal{F}_D\) be the set of all functions of type \(D \to D\). Then, \(\mathcal{F}_D\) is also a complete partial order with the following ordering relation:

\[
f \sqsubseteqD g \iff \forall x \in D. \ f(x) \sqsubseteq g(x)\]
Proof. It is not difficult to prove that $\sqsubseteq^F_D$ is a partial order relation. Its minimal element is the function that maps all elements of $D$ to $\bot$. It remains to prove that every directed subset $M \subseteq \mathcal{F}_D$ has a supremum in $D$. It is easily seen that this supremum is the function $h(x) := \sup\{f(x) \mid f \in M\}$: we clearly have $f \sqsubseteq^F_D h$ for all $f \in M$ if $h$ should exist, and that $h$ is also the least upper bound. Its existence is seen as follows: the set $V_x := \{f(x) \mid f \in M\}$ is a directed subset of $D$, since two elements $f(x)$ and $g(x)$ of it must have an upper bound $g'(x)$ in $V_x$ since $f$ and $g$ must have an upper bound $g'$ in $M$ (since $M$ is a directed subset of $\mathcal{F}_D$).

A.1.3 Fixpoint Theorems

We now consider interesting subsets of functions $f : D \to D$ that have fixpoints. In particular, we also will see how fixpoints of continuous functions can be computed. The next lemma is a first step in this direction:

Definition A.15 (Inflationary Function). A function $f : D \to D$ on a partial order $(D, \sqsubseteq)$ is called inflationary if $x \sqsubseteq f(x)$ holds for all $x \in D$.

Hence, a function $f : D \to D$ is inflationary, if all elements in $D$ are post-fixpoints of $f$. It is interesting to see that all inflationary monotone functions have fixpoints, and they even share a common fixpoint:

Lemma A.16 (Common Fixpoint of Montone Inflationary Functions). Let $(D, \sqsubseteq)$ be a complete partial order and let $I_D$ be the set of monotone and inflationary functions of $\mathcal{F}_D$. Then, the functions $f \in I_D$ share a common fixpoint.

Proof. Using the complete partial order $\sqsubseteq^F_D$ on functions of type $D \to D$, we will prove that $I_D$ is a directed subset of $\mathcal{F}_D$. To this end, we first prove that $I_D$ is closed under function composition:

- $x \sqsubseteq y$ implies $g(x) \sqsubseteq g(y)$ by monotonicity of $g$, and thus $f(g(x)) \sqsubseteq f(g(y))$ by monotonicity of $f$, hence, $f \circ g$ is also monotone.
- Since $g$ is inflationary, we have $x \sqsubseteq g(x)$, and thus by monotonicity of $f$, we conclude $f(x) \sqsubseteq f(g(x))$. Since $f$ is also inflationary, we have $x \sqsubseteq f(x)$, and thus, by transitivity of $\sqsubseteq$, it follows $x \sqsubseteq f(g(x))$, so that $f \circ g$ is also inflationary.

Moreover, $f \circ g$ is an upper bound of $\{f, g\}$:

- Since $g$ is inflationary, we have $x \sqsubseteq g(x)$, and by monotonicity of $f$, we conclude $f(x) \sqsubseteq f(g(x))$.
- $g(x) \sqsubseteq f(g(x))$ follows since $f$ is inflationary.

Hence, the set $I_D$ is a directed subset of the complete partial order $(\mathcal{F}_D, \sqsubseteq^F_D)$. Now, define the following function $h : D \to D$:

$$h(x) := \sup\{f(x) \mid f \in I_D\}$$
Note that \( h \) exists since \( \{ f(x) \mid f \in \mathcal{F} \} \) is a directed subset of \( \mathcal{D} \) and \((\mathcal{D}, \sqsubseteq)\) is a complete partial order. It is moreover easily seen that \( h \in \mathcal{I}_D \), and that \( h \) is therefore the maximum of \( \mathcal{I}_D \):

- \( x \sqsubseteq h(x) \) holds since \( x \sqsubseteq f(x) \) holds for any \( f \in \mathcal{I}_D \).
- If \( x \sqsubseteq y \) holds, then \( h(x) \sqsubseteq h(y) \) follows since \( f(x) \sqsubseteq f(y) \) holds for any \( f \in \mathcal{I}_D \), thus \( \sup(\{ f(x) \mid f \in \mathcal{I}_D \}) \sqsubseteq \sup(\{ f(y) \mid f \in \mathcal{I}_D \}) \).

Thus, by definition of \( h \), it is the maximum of \( \mathcal{I}_D \). Now, for any function \( f \in \mathcal{I}_D \), it follows that \( h \sqsubseteq \delta \circ f \circ h \) holds, since \( \delta \circ h \) is an upper bound of both \( f \) and \( h \). Moreover, we have \( f \circ h \sqsubseteq \delta \circ h \), since \( f \circ h \in \mathcal{I}_D \) and \( h \) is the maximum of \( \mathcal{I}_D \). Thus, we conclude by antisymmetry of \( \sqsubseteq \) that \( f \circ h = h \) holds. This means that for all \( x \in \mathcal{D} \), the element \( h(x) \in \mathcal{D} \) is a fixpoint of all functions \( f \in \mathcal{I}_D \).

By the above lemma, we can now easily prove the existence of least fixpoints of monotone functions on a complete partial order:

**Theorem A.17 (Least Fixpoints of Monotone Functions on CPOs).** Every monotone function \( f : \mathcal{D} \to \mathcal{D} \) over a complete partial order \((\mathcal{D}, \sqsubseteq)\) has a least fixpoint.

**Proof.** Consider the set of post-fixpoints \( \text{Post}_f := \{ x \in \mathcal{D} \mid x \sqsubseteq f(x) \} \) of \( f \). Clearly, \( \sqsubseteq \) is also a partial order on \( \text{Post}_f \), and we will moreover prove that \((\text{Post}_f, \sqsubseteq)\) is a complete partial order:

- \( \bot \in \text{Post}_f \) holds, since \( \bot \sqsubseteq f(\bot) \) obviously holds, since \( \bot \) is the minimal element of \( \mathcal{D} \).
- Assume \( M \subseteq \text{Post}_f \) is a directed subset of \( \text{Post}_f \). Then, \( M \) is also a directed subset of \( \mathcal{D} \), and by the completeness of \((\mathcal{D}, \sqsubseteq)\), it follows that \( \sup(M) \in \mathcal{D} \) exists. For the completeness of \((\text{Post}_f, \sqsubseteq)\), we have to prove that even \( \sup(M) \in \text{Post}_f \) holds. This is easily seen as follows: \( \forall x \in M. x \sqsubseteq \sup(M) \) holds by definition of \( \sup(M) \), and since \( f \) is monotone, we also have \( \forall x \in M. f(x) \sqsubseteq f(\sup(M)) \). Since \( M \subseteq \text{Post}_f \), we moreover have \( \forall x \in M. x \sqsubseteq f(x) \sqsubseteq f(\sup(M)) \). Thus, \( f(\sup(M)) \) is also an upper bound of \( M \), and therefore we have \( \sup(M) \sqsubseteq f(\sup(M)) \). This means \( \sup(M) \in \text{Post}_f \), and therefore \((\text{Post}_f, \sqsubseteq)\) is complete.

Therefore, \((\text{Post}_f, \sqsubseteq)\) is a sub-cpo of \((\mathcal{D}, \sqsubseteq)\). Consider now the restriction of \( f \) on \( \text{Post}_f \): for \( x \in \text{Post}_f \), we have \( x \sqsubseteq f(x) \), and thus by monotonicity of \( f \) also \( f(x) \sqsubseteq f(f(x)) \). Hence, we conclude that \( f(x) \in \text{Post}_f \), and therefore, we have \( f_{\text{Post}_f} : \text{Post}_f \to \text{Post}_f \). For this reason, \( f_{\text{Post}_f} : \text{Post}_f \to \text{Post}_f \) is an inflationary monotone function on the cpo \((\text{Post}_f, \sqsubseteq)\), and by Lemma A.16, it follows that \( f_{\text{Post}_f} \) has a fixpoint in \( \text{Post}_f \). Therefore, \( f \) has at least one fixpoint in \( \mathcal{D} \), so that \( \text{Fix}_f := \{ x \in \mathcal{D} \mid f(x) = x \} \) is not empty.

Similar to \( \text{Post}_f \), we note that for any fixpoint \( x_f \in \text{Fix}_f \), the set \( \text{Down}(x_f) := \{ x \in \mathcal{D} \mid x \sqsubseteq x_f \} \) is also a sub-cpo of \((\mathcal{D}, \sqsubseteq)\) where the restriction \( f_{\text{Down}(x_f)} \) of \( f \) is an inflationary function:
• \( \bot \in \text{Down}(x_f) \) holds, since \( \bot \subseteq x_f \) obviously holds.

• Assume \( M \subseteq \text{Down}(x_f) \) is a directed subset of \( \text{Down}(x_f) \). Then, \( M \) is also a directed subset of \( D \), and by the completeness of \( D \), it follows that \( \sup(M) \in D \) exists. Since \( x_f \) is an upper bound of \( \text{Down}(x_f) \), we clearly have \( \sup(M) \subseteq x_f \). Thus \( \sup(M) \in \text{Down}(x_f) \) holds, and therefore \( (\text{Down}(x_f), \sqsubseteq) \) is complete.

• Moreover, we have \( f(\text{Down}(x_f)) \subseteq \text{Down}(x_f) \), since for \( x \in \text{Down}(x_f) \), we have \( x \subseteq x_f \), and thus by monotonicity of \( f \) also \( f(x) \sqsubseteq f(x_f) = x_f \), so that \( f(x) \in \text{Down}(x_f) \) follows.

For this reason, the set \( \text{Down}(x_f) := \{ x \in D \mid x \subseteq x_f \} \) is for any \( x_f \in \text{Fix}_f \) a sub-cpo of \( (D, \sqsubseteq) \), and \( f_{\text{Down}(x_f)} : \text{Down}(x_f) \rightarrow \text{Down}(x_f) \) is an inflationary monotone function on \( \text{Down}(x_f) \). Now consider the intersection \( D_0 \) of all \( \text{Down}(x_f) \), i.e.,

\[
D_0 := \bigcap_{x_f \in \text{Fix}_f} \text{Down}(x_f)
\]

Similar to each \( \text{Down}(x_f) \), we prove that \( D_0 \) is also a sub-cpo of \( (D, \sqsubseteq) \) where the restriction \( f_{|D_0} \) of \( f \) is an inflationary function:

• \( \bot \in D_0 \) holds, since \( \bot \in \text{Down}(x_f) \) for every \( x_f \).

• Assume \( M \subseteq D_0 \) is a directed subset of \( D_0 \), and therefore also a directed subset of each \( \text{Down}(x_f) \) for every \( x_f \in \text{Fix}_f \). We prove now that one and the same element \( s_M \in D \) is the supremum of \( M \) in \( D_0 \) and in each \( \text{Down}(x_f) \): To this end, consider an arbitrary \( x_f \in \text{Fix}_f \). Since \( M \subseteq D_0 \subseteq \text{Down}(x_f) \) holds, it follows that each \( x_f \) is an upper bound of \( M \), and therefore, we have \( s_M \subseteq x_f \), which implies \( s_M \in \text{Down}(x_f) \). As \( s_M \) belongs to every \( \text{Down}(x_f) \), it also belongs to \( D_0 \), and thus, \( (D_0, \sqsubseteq) \) is complete.

• Finally, assume \( x \in D_0 \). Then, we have \( x \in \text{Down}(x_f) \) for every \( x_f \in \text{Fix}_f \), and therefore \( x \subseteq x_f \) for every \( x_f \in \text{Fix}_f \), and by monotonicity of \( f \), also \( f(x) \subseteq f(x_f) = x_f \), so that \( f(x) \in \text{Down}(x_f) \) follows for every \( x_f \in \text{Fix}_f \). Hence, we have \( f(x) \in D_0 \).

Hence, \( f \) does also have a fixpoint in \( D_0 \). We now prove that there is no more than one fixpoint in \( D_0 \): Assume we have fixpoints \( x_1, x_2 \in D_0 \) with \( x_1 \neq x_2 \). Then, \( x_2 \notin \text{Down}(x_1) \) follows since otherwise, \( D_0 \subseteq \text{Down}(x_2) \subseteq \text{Down}(x_1) \) would hold, which would lead to the contradiction \( x_1 \notin D_0 \). However, \( x_2 \notin \text{Down}(x_1) \) leads to the contradiction \( x_2 \notin D_0 \), since \( D_0 \subseteq \text{Down}(x_1) \) holds.

Thus, there is exactly one fixpoint of \( f \) in \( D_0 \), and by construction of \( D_0 \), it is easily seen that this is the least fixpoint of \( f \) (it is contained in all sets \( \text{Down}(x_f) \), and thus, is less than or equal to all other fixpoints).

Note that a directed subset may have different suprema in different sub-cpos of a cpo. For example, consider the set \( D := \mathbb{N} \cup \{ a, b \} \) where for all \( m, n \in \mathbb{N} \), we have \( n \subseteq m \iff m \leq n, a \subseteq n, \) and \( b \subseteq n \). It follows that \( (D, \sqsubseteq) \) is not a complete partial order, since the upper bounds of \( \{ a, b \} \) are \( \mathbb{N} \), and
hence, there is no least upper bound. However, every set \( \{a, b, 0, \ldots, n\} \) is a complete partial order, and the upper bounds of \( \{a, b\} \) are \( \{0, \ldots, n\} \), so that \( n \) is the least upper bound. This is, however, a different element in each sub-cpo \( \{a, b, 0, \ldots, n\} \).

By the above theorem, we already know that if a partial order is complete, then every monotone function has a least fixpoint. It is known that the converse is also true:

**Theorem A.18.** A partial order \( (D, \sqsubseteq) \) is complete iff every monotone function \( f : D \to D \) has a least fixpoint.

The proof of the above theorem is however not that simple, and since we do not make use of the above theorem, we omit it here (the interested reader is referred to [75, 219]). Moreover, the proof of Theorem A.17 does not lead to an algorithm, so that we have so far no means to compute the least fixpoint of a monotone function. However, if the function is even continuous, then there is a simple algorithm for computing (or at least approximating) fixpoints:

**Theorem A.19 (Kleene’s Fixpoint Theorem (I)).** Let \( (D, \sqsubseteq) \) be a complete partial order and \( f : D \to D \) be a continuous function. Then, the sequence \( x_0 := \bot \) and \( x_{i+1} := f(x_i) \) converges to the least fixpoint of \( f \) (which exists by Theorem A.17).

**Proof.** In the proof of Theorem A.17, we have already proved that the sequence \( x_i \) is monotone, i.e., (1) \( \forall n. x_n \sqsubseteq x_{n+1} \) holds. Therefore, the set \( C := \{f^n(\bot) \mid n \in \mathbb{N}\} \) is totally ordered, and therefore, it is a directed subset of \( D \). Since our partial order is complete, it follows that \( C \) has a supremum \( \sup(C) \). Moreover, by continuity of \( f \), it follows that (2) \( f(\sup(C)) = \sup(f(C)) \).

We next observe that (3) \( \sup(C) = \sup(f(C)) \), since \( f(C) = \{x_1, x_2, \ldots\} \) and \( \bot = x_0 \sqsubseteq x_i \). Thus, we conclude by (2) and (3) that (4) \( f(\sup(C)) = \sup(C) \), i.e., \( \sup(C) \) is a fixpoint of \( f \).

As we now know that \( f \) has fixpoints, it is now reasonable to consider the set of its fixpoints \( \mathcal{F}_f := \{x \in D \mid f(x) = x\} \). We already know that \( \mathcal{F}_f \) is not empty, since at least \( \sup(C) \in \mathcal{F}_f \).

We now prove by induction on \( n \) that (5) \( \forall n. f^n(\bot) \subseteq x_f \) for every \( x_f \in \mathcal{F}_f \). The induction base is trivial, since \( f^0(\bot) = \bot \subseteq x_f \) for all \( x_f \in \mathcal{F}_f \). In the induction step, we know by the induction hypothesis that \( f^n(\bot) \subseteq x_f \) holds for all \( x_f \in \mathcal{F}_f \). By monotonicity of \( f \), it therefore follows \( f^{n+1}(\bot) \subseteq f(x_f) = x_f \) for all \( x_f \in \mathcal{F}_f \), so that the induction step is proved.

Hence, (5) holds, and thus, all fixpoints \( x_f \in \mathcal{F}_f \) are upper bounds of \( C \). By definition of the supremum, it therefore follows that \( \sup(C) \) is the least fixpoint of \( f \), i.e., \( \sup(C) = \min(\mathcal{F}_f) \).

The above theorem can be generalized in some ways. The most important improvement is that the sequence \( x_i \) may start with an arbitrary post-fixpoint \( x_0 \) less than or equal to the fixpoint \( x_f \):
Theorem A.20 (Kleene’s Fixpoint Theorem (II)). Let $(D, \sqsubseteq)$ be a complete partial order and $f : D \to D$ be a continuous function whose least fixpoint is $x_f$. Moreover, let $x_0 \in D$ be such that $(\mu_1) x_0 \sqsubseteq f(x_0)$ and $(\mu_2) x_0 \sqsubseteq x_f$. Then, the sequence $x_{i+1} := f(x_i)$ starting with $x_0$ converges to the least fixpoint $x_f$ of $f$.

Proof. The proof is essentially the same as for the previous theorem. We first prove again that the sequence $x_i$ is monotone, i.e., that $\forall n.x_n \sqsubseteq x_{n+1}$ holds. For $n = 0$ this follows by assumption $(\mu_1)$. The induction step is proved exactly as in proof of the previous theorem.

Thus, $C := \{ f^n(x_0) \mid n \in \mathbb{N} \}$ is a chain so that its supremum $\sup(C)$ exists, and by the same arguments as in the proof of the previous theorem, we see that $\sup(C)$ is a fixpoint of $f$.

Next, consider again the set of fixpoints $F_f := \{ x \in D \mid f(x) = x \}$ of $f$. We now prove by induction on $n$ that $\forall n.f^n(x_0) \sqsubseteq x_f$ for every $x_f \in F_f$. The induction base follows this time by assumption $(\mu_2)$, while the induction step is proved as in the proof of the previous theorem. The rest is exactly the same as in the proof of the previous theorem.

The previous two fixpoint theorems are the most important ones for complete partial orders. There are also others, e.g. one can prove that every inflationary function $f : D \to D$ does have a least fixpoint [219].

One might wonder whether the limit of the sequence defined a $x_0 := \bot$ and $x_{i+1} := f(x_i)$ does converge even to fixpoint even if the function is only monotone, but not continuous. We will prove below that it does at least converge to post-fixpoint, but the counterexample given after the proof of the next lemma shows that it will not necessarily converge to a fixpoint.

Lemma A.21 (Post-Fixpoints of Monotone Functions on CPOs). Let $f : D \to D$ be a monotone function over a complete partial order $(D, \sqsubseteq)$, and define the sequence $x_0 := \bot$ and $x_{i+1} := f(x_i)$ as well as the set $C_f := \{ f^i(\bot) \mid i \in \mathbb{N} \}$. Then, $C_f$ is a chain, and $\sup(C_f)$ is a post-fixpoint of $f$.

Proof. We first prove that the sequence $x_i$ is monotone, i.e., that (1) $\forall i.x_i \sqsubseteq x_{i+1}$ holds. For $i = 0$, this follows the definition of $x_0 := \bot$. By monotonicity of $f$, we derive from the induction hypothesis $x_i \sqsubseteq x_{i+1}$ and the monotonicity of $f$ that $f(x_i) \sqsubseteq f(x_{i+1})$ holds, i.e., that $x_{i+1} \sqsubseteq x_{i+2}$ holds. Hence, (1) holds, i.e., the sequence $x_i$ is monotonically growing, and therefore $C_f$ is a chain. By completeness of $(D, \sqsubseteq)$, we therefore know that $\sup(C_f)$ exists.

We consider now the set of pre-fixpoints $\text{Pre}_f := \{ x \in D \mid f(x) \sqsubseteq x \}$ of $f$ and prove by induction on $i$ that (2) $\forall i.\forall x \in \text{Pre}_f. f^i(\bot) \sqsubseteq x$. The induction base is trivial, since $f^0(\bot) = \bot \sqsubseteq x$ for all $x \in D$, and hence also for all $x \in \text{Pre}_f$. In the induction step, we already know by the induction hypothesis that $\forall x \in \text{Pre}_f. f^i(\bot) \sqsubseteq x$ holds, and since $f$ is monotone, we conclude that $\forall x \in \text{Pre}_f. f^{i+1}(\bot) \sqsubseteq f(x)$. Since all $x \in \text{Pre}_f$ have the property $f(x) \sqsubseteq x$, we conclude by transitivity of $\sqsubseteq$ that $\forall x \in \text{Pre}_f. f^{i+1}(\bot) \sqsubseteq x$ holds. Thus, we have (2) $\forall i.\forall x \in \text{Pre}_f. f^i(\bot) \sqsubseteq x$.
Since by (2) all elements \( x \) in \( \text{Pre}_f \) are upper bounds of \( C_f \), and \( \sup(C_f) \) is the least upper bound of \( C_f \), we also have (3) \( \forall x \in \text{Pre}_f, \sup(C_f) \subseteq x \).

From (2), we obtain \( \forall i, f^i(\bot) \subseteq \sup(C_f) \), and by monotonicity of \( f \) that also \( \forall i. f^i(\bot) \subseteq f(\sup(C_f)) \) holds (the additional \( f^0(\bot) = \bot \subseteq f(\sup(C_f)) \) is trivial). Hence, \( f(\sup(C_f)) \) is also an upper bound of \( C_f \), and since \( \sup(C_f) \) is the least upper bound, we have (4) \( \sup(C_f) \subseteq f(\sup(C_f)) \), so that \( \sup(C_f) \) is a post-fixpoint of \( f \).

As however \( \bot \) is post-fixpoint of every function, it is not interesting to compute post-fixpoints of a function. It is interesting to see that \( \sup(\{f^i(\bot) \mid i \in \mathbb{N}\}) \) used in the proof of Lemma A.21 may not be a fixpoint of the monotone function \( f \). To this end, consider the set \( \mathbb{N} \times \mathbb{N} \) with the following total order relation:

\[
(m_1, n_1) \triangleq (m_2, n_2) \iff \begin{cases} n_1 \leq n_2 & : \text{if } m_1 = m_2 \\
_1 \leq n_2 & : \text{otherwise}
\end{cases}
\]

It is easily seen that \( (\mathbb{N} \times \mathbb{N}, \subseteq) \) is a total order with least element \( \bot = (0, 0) \). However, it is not complete, since the entire set \( \mathbb{N} \times \mathbb{N} \) is a chain, but has no supremum. For this reason, we add a new element \( \top \) and extend the order relation such that \( x \subseteq \top \) holds for all elements \( x \in (\mathbb{N} \times \mathbb{N}) \cup \{\top\} \). Then, \( (\mathbb{N} \times \mathbb{N}) \cup \{\top\} \) is a complete total order.

Define the function \( f \) as \( f((m, n)) := (m, n + 1) \) and \( f(\top) = \top \). Hence, \( f \) is monotone, and its single fixpoint is \( \top \). We obtain \( \hat{x} := \sup(\{f^i((0, 0)) \mid i \in \mathbb{N}\}) = \sup((0, i) \mid i \in \mathbb{N}) = (1, 0) \). However, since \( f((1, 0)) = (1, 1) \neq (1, 0) \) holds, we see that \( \hat{x} \) is not a fixpoint of \( f \). Note that \( f \) is monotone, but not continuous, since \( f(\sup((0, i) \mid i \in \mathbb{N})) = f((1, 0)) = (1, 1) \neq \sup(f((0, i) \mid i \in \mathbb{N})) = (1, 0) \).

Hence, \( \sup(\{f^i(\bot) \mid i \in \mathbb{N}\}) \) may not be a fixpoint for a monotone function in a cpo. However, it can be proved that the transfinite\(^1 \) sequence \( x_0 := \bot \), and \( x_{i+1} := f(x_i) \) converges to a fixpoint.

### A.1.4 Cartesian Products of Partial Orders

For many applications, one has to consider compound partial orders, i.e., partial orders that are obtained from other partial orders. In particular, the cartesian product of partial orders is thereby of interest that is defined as follows:

**Theorem A.22 (Products of Complete Partial Orders).**

Given partial orders \((D_1, \subseteq_1), \ldots, (D_n, \subseteq_n)\), then the cartesian product \(D_1 \times \ldots \times D_n\), endowed with the following relation is also a partial order:

\[
(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \iff \bigwedge_{i=1}^n x_i \sqsubseteq_i y_i
\]

\(^1\)‘Transfinite’ means that the indices \( i \) range over all ordinal numbers (and not only over the natural numbers).
Moreover, if the partial orders \((D_i, \sqsubseteq_i)\) are all complete, then also the above product is a complete partial order.

**Proof.** It is easily seen that \(\sqsubseteq\) is a partial order. Its completeness is also easily seen: Every directed subset \(M = M_1 \times \ldots \times M_n\) of \(D_1 \times \ldots \times D_n\) consists of directed subsets \(M_i\) of \(D_i\). Since the partial orders \((D_i, \sqsubseteq_i)\) are all complete, each \(M_i\) has a supremum \(s_i \in D_i\). Then, the supremum of \(M\) is \((s_1, \ldots, s_n)\). The least element is certainly the tuple of least elements of the component cpos \(D_i\).

Hence, the computation of fixpoints of continuous functions \(f : D_1 \times D_2 \to D_1 \times D_2\) can be done by the Tarski/Knaster iteration. The following lemma shows that we can reduce this fixpoint computation to others of functions of types \(D_1 \to D_1\) and \(D_2 \to D_2\).

**Lemma A.23 (Bekič’s Lemma on Simultaneous Fixpoints [26]).** Given complete partial orders \((D_1, \sqsubseteq_1)\) and \((D_2, \sqsubseteq_2)\), and two continuous functions \(f : D_1 \times D_2 \to D_1\) and \(g : D_1 \times D_2 \to D_2\), we define the function \(h : D_1 \times D_2 \to D_1 \times D_2\) as \(h(x, y) := (f(x, y), g(x, y))\). Then, \(h\) is continuous and the least fixpoint \((\bar{x}, \bar{y})\) of \(h\) is determined as \(\bar{x} := \mu x.f(x, \mu y.g(x, y))\) and \(\bar{y} := \mu y.g(\bar{x}, y)\). Equivalently, the least fixpoint of \(h\) can be described as \((\bar{x}', \bar{y}')\) with \(\bar{x}' := \mu x.f(x, \bar{y}')\) and \(\bar{y}' := \mu y.g(\bar{x}', y)\).

**Proof.** Clearly, \(h\) is continuous, and therefore \(h\) has fixpoints, in particular, a least fixpoint. Let \((x_0, y_0)\) be the least fixpoint of \(h\). For convenience, we define the function \(g' : D_1 \to D_2\) as \(g'(x) := \mu y.g(x, y)\). Then, the following facts immediately follow by the definitions of \((x_0, y_0)\), \(\bar{x}\), \(\bar{y}\), and \(g'\):

1. \(\forall x.g'(x) = g(x, g'(x))\)
2. \(\forall x.\forall y.g(x, y) \sqsubseteq_2 y \Rightarrow g'(x) \sqsubseteq_2 y\)
3. \(\bar{y} = g'(\bar{x})\)
4. \(\bar{x} = f(\bar{x}, g'(\bar{x}))\)
5. \(\forall x.f(x, g'(x)) \sqsubseteq_1 x \Rightarrow \bar{x} \sqsubseteq_1 x\)
6. \(x_0 = f(x_0, y_0)\)
7. \(y_0 = g(x_0, y_0)\)
8. \(\forall x.\forall y.f(x, y) \sqsubseteq_1 x \land g(x, y) \sqsubseteq_2 y \Rightarrow x_0 \sqsubseteq_1 \bar{x} \land y_0 \sqsubseteq_2 \bar{y}\)

(1) and (2) are due to the definition of \(g'\), since \(g'(x)\) is the least fixpoint of \(\lambda y.g(x, y)\). (3) is the definition of \(\bar{y}\) rewritten with the definition of \(g'\). (4) and (5) are the definition of \(\bar{x}\), where we also used the function \(g'\) for conciseness. (6), (7), and (8) formalize that \((x_0, y_0)\) is the least fixpoint of \(h\).

By (3) and (4) we immediately obtain (9) \(\bar{x} = f(\bar{x}, \bar{y})\) and from (3) and (9) that (10) \(\bar{y} = g(\bar{x}, \bar{y})\) holds. Hence, \((\bar{x}, \bar{y})\) is a fixpoint of \(h\). Therefore, we obtain by modus ponens from (9), (10), and (8) that (11) \(x_0 \sqsubseteq_1 \bar{x}\) and (12) \(y_0 \sqsubseteq_2 \bar{y}\) holds.
From (7), we get \( g(x_0, y_0) \sqsubseteq y_0 \), hence, it follows by modus ponens with (2) that (13) holds. By monotonicity of \( f \), we therefore obtain \( f(x_0, g(x_0)) \sqsubseteq f(x_0, y_0) \), which means by (6) that \( f(x_0, g(x_0)) \sqsubseteq x_0 \) holds. Modus ponens with (5) therefore shows us (14) holds.

Using (14) and the monotonicity of \( g \), we now obtain \( g(x_0, y_0) \sqsubseteq g(x_0, y_0) \), and considering (7) simplifies this to \( g(x, y_0) \sqsubseteq y_0 \). Modus ponens with (2) results in \( g(x) \sqsubseteq y_0 \), which means by (3) that (15) holds.

Summing up, by (11) and (14), it follows that \( x = x_0 \) and by (12) and (15), it follows that \( y = y_0 \) holds. Hence, \( (x, y) \) is the least fixpoint of \( h \).

The second characterization of the least fixpoint of \( h \) is obtained as follows: Consider \( h'(y, x) := (g(x, y), f(x, y)) \). By the proven facts, \( h' \) has the least fixpoint \( (\hat{y}', \hat{x}') \) where \( \hat{y}' \) and \( \hat{x}' \) are as described in the lemma. It is however obvious that \( h(x, y) = (a, b) \) iff \( h'(y, x) = (b, a) \) hold.

The above lemma is important for the understanding of simultaneous fixpoint definitions. It moreover has some surprising consequences that we will consider in the following. For example, note that we have

\[
\hat{x} := \mu y. f(x, \mu y. g(x, y)) = \mu z. f(z, \mu y. g(\mu x. f(x, y), y)) := \hat{x'},
\]

since least fixpoints are uniquely determined. In the special case where \( f(x, y) := y \) holds, this reduces to \( \mu x. \mu y. g(x, y) = \mu y. g(f(y), y) \), so that we see multiple least fixpoint operators that follow on each other can be merged to a single one.

A further result that is more important is that we can ‘serialize’ the computation of simultaneous fixpoints, i.e., we can reduce it to the computation of some fixpoints that are not simultaneous. This is very important for the construction of model checking procedures for the vectorized \( \mu \)-calculus [225].

**Lemma A.24 (Special Fixpoint Iterations).** Given complete partial orders \((D_1, \sqsubseteq_1)\) and \((D_2, \sqsubseteq_2)\) and the function \( h : D_1 \times D_2 \to D_1 \times D_2 \) defined by two continuous functions \( f : D_2 \to D_1 \) and \( g : D_1 \times D_2 \to D_2 \) as \( h(x, y) := (f(y), g(x, y)) \). If \( h \) is continuous, then the least fixpoint of \( h \) is \( (f(\hat{y}), \hat{y}) \) with \( \hat{y} := \mu y. g(f(y), y) \). Moreover, the following two iteration schemes both compute this fixpoint:

\[
\begin{align*}
  x_0 &:= \bot \\
  y_0 &:= \bot \\
  x_{i+1} &:= f(y_i) \\
  y_{i+1} &:= g(x_{i+1}, y_i)
\end{align*}
\]

\[
\begin{align*}
  x_0 &:= \bot \\
  y_0 &:= \bot \\
  x_{i+1} &:= f(y_i) \\
  y_{i+1} &:= \mu y. g(x_{i+1}, y)
\end{align*}
\]

**Proof.** According to Lemma A.23, the least fixpoint of \( h \) is given as \( (f(\hat{y}), \hat{y}) \) where \( \hat{y} := \mu y. g(f(y), y) \) holds. Hence, we can compute \( \hat{y} \) according to the Tarski/Knaster theorem by the iteration sequence: \( y_0 := \bot, y_{i+1} := g(f(y_i), y_i) \). Abbreviating \( x_0 := \bot \) and \( x_{i+1} := f(y_i) \) transforms this iteration directly into the one that is given on the left hand side of the lemma. Note further that \( \lim_{i \to \infty} x_i = \lim_{i \to \infty} x_{i+1} = \lim_{i \to \infty} f(y_i) = f(\lim_{i \to \infty} y_i) = f(\hat{y}) \) holds, thus, the sequence \((x_i, y_i)\) converges to the least fixpoint \((f(\hat{y}), \hat{y})\) of \( h \).
Now consider the iteration given on the right hand side of the lemma. We replace \( x_{i+1} \) by \( f(y_i) \) in the iteration for \( y_{i+1} \) and therefore see that \( \lim_{i \to \infty} y_i = \mu z. \mu y. g(f(z), y) \). To see this, consider the Tarski iteration that is obtained for this fixpoint: \( z_0 := \bot \) and \( z_{i+1} := \mu y. g(f(z_i), y) = \mu y. g(x_{i+1}, y) \).

We have already seen that a consequence of Lemma A.23 is, that we can merge nested least fixpoint operators. Therefore, we see that \( \mu z. \mu y. g(f(z), y) = \mu y. g(f(y), y) = \check{y} \) holds. Hence, \( y_i \) converges to \( \check{y} \) and therefore, by the same argument as before, \( x_i \) converges to \( f(\check{y}) \), so that the iteration also computes the least fixpoint \( (f(\check{y}), \check{y}) \) of \( h \).

The above iterations are used in model checking algorithms for the modal \( \mu \)-calculus, but of course, they can also be used for any fixpoint computations of simultaneous fixpoints, e.g. for an efficient causality analysis.

A.2 Complete Lattices

The previous section considered complete partial orders and presented conditions that guarantee the existence of fixpoints of functions \( f : \mathcal{D} \to \mathcal{D} \). In this section, we strengthen these results to lattices and complete lattices.

A.2.1 Lattices

Lattices can be defined in two ways: either via a characterization by order theory or by algebraic properties. We proceed by an order-theoretic definition and state the second characterization as a theorem.

**Definition A.25 (Lattices).** A partially ordered set \( (\mathcal{D}, \sqsubseteq) \) is a lattice, if all sets \( \{x, y\} \subseteq \mathcal{D} \) have \( \sup(\{x, y\}) \) and \( \inf(\{x, y\}) \) in \( \mathcal{D} \). In a lattice \( (\mathcal{D}, \sqsubseteq) \), we can therefore define the following two binary operations (called meet and join):

- \( x \sqcap y := \sup(\{x, y\}) \)
- \( x \sqcup y := \inf(\{x, y\}) \)

An equivalent definition of lattices is obtained by replacing two-element sets \( \{x, y\} \) by arbitrary nonempty finite sets \( M_n \subseteq \mathcal{D} \) in the above definition. Note that the difference between directed subsets of a partial order and lattices is not just that also lower bounds are demanded. Lattices do not only demand that two-element sets have an upper bound, they must also have a supremum. The following example shows the difference:

\( \{a, b, c, d, e\} \) is a partial order, however, it is not a lattice, since e.g. the set \( \{d, e\} \) has upper bounds \( \{a, b, c\} \), but it does not have a least upper bound. However, \( \{a, b, c, d, e\} \) is a directed subset of itself. Hence, complete partial orders, lattices, and directed subsets are rather different things.

As already noted above, lattices also have an equivalent algebraic definition:
Fig. A.1. Example of a directed set that is even a complete partial order, but not a lattice.

**Theorem A.26 (Algebraic Properties of Lattices).** In every lattice \((D, \sqsubseteq)\), the following equations hold:

- **Commutative Laws:**
  \[ x \sqcap y = y \sqcap x \]
  \[ x \sqcup y = y \sqcup x \]

- **Associative Laws:**
  \[ x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \]
  \[ x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z \]

- **Absorption Laws:**
  \[ x \sqcap (x \sqcup y) = x \]
  \[ x \sqcup (x \sqcap y) = x \]

- **Idempotency Laws:**
  \[ x \sqcap x = x \]
  \[ x \sqcup x = x \]

It should be remarked that distributive laws need, in general, not hold. Furthermore, if a greatest element \(\top\) and a least element \(\bot\) exist, then we have \(x \sqcap \top = x\), \(x \sqcup \bot = \bot\), \(x \sqcup \top = \top\), and \(x \sqcap \bot = x\). We also have the converse of the above theorem:

**Theorem A.27 (Algebraic Properties of Lattices).** Let \(D\) be a set with two binary operations \(\sqcap\) and \(\sqcup\) that satisfy the commutative laws, the associative laws, and the absorption laws. Then, the relation \(\sqsubseteq\) defined as \(x \sqsubseteq y \iff x = x \sqcap y\) is a partial order that makes \(D\) a lattice, such that \(x \sqcup y = \sup\{x, y\}\) and \(x \sqcap y = \inf\{x, y\}\) holds.

We omitted the idempotency laws since they can be derived from the absorption laws. Hence, there are two alternative definitions of lattices, one by algebraic properties, and another by partial orders.

For finite sets \(M\), \(\inf(M)\) and \(\sup(M)\) can be recursively determined in a lattice by just requiring to compute \(\inf\{x, y\}\) and \(\sup\{x, y\}\) for two-element sets:

**Lemma A.28 (Properties of Lattices).** In every lattice \((D, \sqsubseteq)\), we have the following laws:
A.2 Complete Lattices

- \( \inf\{e_1, \ldots, e_{n+1}\} = \inf(\{e_{n+1}, \inf(\{e_1, \ldots, e_n\})\}) \)
- \( \sup(\{e_1, \ldots, e_{n+1}\}) = \sup(\{e_{n+1}, \sup(\{e_1, \ldots, e_n\})\}) \)

and therefore, \( \inf(M) \) and \( \sup(M) \) exist for every finite subset \( M \subseteq D \) in every lattice.

Hence, in a lattice, all finite sets \( M \subseteq D \) have \( \inf(M) \) and \( \sup(M) \) in \( D \).

A.2.2 Complete Lattices

In the previous subsection, we considered lattices that demand that \( \sup(M) \) and \( \inf(M) \) exist for all two-element sets \( M \), or alternatively for all finite sets \( M \). The complete partial orders as considered in the previous section demand that all directed subsets \( M \) have a supremum \( \sup(M) \). The complete lattices as considered in this subsection are further restrictions that are defined as follows:

**Definition A.29 (Complete Lattices).** A partially ordered set \((D, \sqsubseteq)\) is a complete lattice, if for every set \( M \subseteq D \), \( \sup(M) \) and \( \inf(M) \) exist in \( D \). In particular, we write \( \bot := \inf(D) \) and \( \top := \sup(D) \).

Compared to lattices, the additional requirement of complete lattices is that \( \sup(M) \) and \( \inf(M) \) must also exist for infinite sets \( M \). To see that there are lattices that are not complete, note that \((\mathbb{N}, \leq)\) is a lattice (since every total order is a lattice), but it is not a complete lattice (since \(\mathbb{N}\) has no supremum).

If we add a new element \( \top \) and define \( n \sqsubseteq m \iff n \leq m \) for \( n, m \in \mathbb{N} \), and \( n \sqsubseteq \top \) for every \( n \in \mathbb{N} \), then \((\mathbb{N} \cup \{\top\}, \sqsubseteq)\) becomes a complete lattice. By Lemma A.28, we can moreover conclude the following theorem:

**Theorem A.30 (Finite Lattices).** Every finite lattice is complete.

A complete lattice is not only a lattice, it is also a complete partial order. The converse is in general not the case: For example, the complete partial order of streams \((D^\omega, \sqsubseteq)\) with the prefix relation \( \sqsubseteq \) is not a complete lattice, since only its directed subsets have suprema, while even a two-element set consisting of two different infinite streams has not even an upper bound. Another, and even finite example is shown in Figure A.1. However, we can establish the following relationship to complete partial orders:

**Theorem A.31 (Complete Partial Orders and Complete Lattices).** For every partial order \((D, \sqsubseteq)\), the pair \((D, \sqsubseteq^{-1})\) with \( x \sqsubseteq^{-1} y :\iff y \sqsubseteq x \) is also a partial order. Moreover, if \((D, \sqsubseteq)\) is a complete lattice then both \((D, \sqsubseteq)\) and \((D, \sqsubseteq^{-1})\) are complete partial orders.

**Proof.** It is easily seen that \((D, \sqsubseteq^{-1})\) is a partial order if \((D, \sqsubseteq)\) is a partial order. If \((D, \sqsubseteq)\) is a complete lattice, then every set \( M \subseteq D \) has a supremum and an infimum in \( D \). Thus, every directed subset \( M \subseteq D \) has a supremum in \( D \) wrt. \( \sqsubseteq \). The infimum of \( M \) wrt. to \( \sqsubseteq \) is the supremum of \( M \) wrt. \( \sqsubseteq^{-1} \), so that also \((D, \sqsubseteq^{-1})\) is complete. Moreover, the minimal element of \((D, \sqsubseteq^{-1})\) is \( \top \), while the minimal element of \((D, \sqsubseteq)\) is clearly \( \bot \). \( \square \)
By symmetric arguments, we conclude that $\text{Fix}$, thus, by $\max$ which moreover satisfy $\max(\text{Fix}_f)$ is even a complete lattice. In particular, there is even a least and a greatest fixpoint $\min(\text{Fix}_f)$.

We can therefore make use of the already mentioned properties like the fixpoint theorem of complete partial orders even for both $(D, \sqsubseteq)$ and $(D, \sqsubseteq^{-1})$. A first result is therefore that in complete lattices, monotone functions $f : D \to D$ have a least and a greatest fixpoint [153, 251]:

**Theorem A.32 (Fixpoints of Monotone Functions (Tarski/Knaster (I))).**

Every monotone function $f : D \to D$ on a complete lattice $(D, \sqsubseteq)$ has fixpoints, i.e., elements $x \in D$ such that $f(x) = x$ holds, and the set of its fixpoints $\text{Fix}_f$ is even a complete lattice. In particular, there is even a least and a greatest fixpoint which moreover satisfy $\max(\text{Fix}_f) = \max(\text{Post}_f)$ and $\min(\text{Fix}_f) = \min(\text{Pre}_f)$ with $\text{Pre}_f := \{ x \in D \mid f(x) \sqsubseteq x \}$ and $\text{Post}_f := \{ x \in D \mid x \sqsubseteq f(x) \}$.

**Proof.** We follow Tarski’s original proof [251]. Define (1) $u := \sup(\text{Post}_f) := \sup(\{ x \in D \mid x \sqsubseteq f(x) \})$, which exists since $(D, \sqsubseteq)$ is a complete lattice, and $\text{Post}_f \neq \emptyset$ since $\bot \in \text{Post}_f$. For every $x \in \text{Post}_f$, we have $x \sqsubseteq f(x)$, and by definition of $u$ therefore $x \sqsubseteq u$. By monotonicity of $f$, we therefore have $f(x) \sqsubseteq f(u)$, and thus $x \sqsubseteq f(u)$ for all $x \in \text{Post}_f$ (remember $x \sqsubseteq f(x)$), so that $f(u)$ is also an upper bound of $\text{Post}_f$. Since $u$ is the least upper bound of $\text{Post}_f$, we therefore conclude that (2) $u \sqsubseteq f(u)$ holds. By monotonicity of $f$, we furthermore see $f(u) \sqsubseteq f(f(u))$, so that $f(u) \in \text{Post}_f$ follows. For this reason, we conclude by definition of $u$ that (3) $f(u) \sqsubseteq u$ must hold, and thus, by (2) and (3) that $f(u) = u$ holds. Therefore, $u$ is a fixpoint of $f$.

Since $\text{Fix}_f \subseteq \text{Post}_f$ holds, we see that (**) $\min(\text{Fix}_f) = \min(\text{Pre}_f)$, and $\max(\text{Fix}_f)$ exist. By symmetric arguments, we conclude that (**) $\min(\text{Fix}_f) = \min(\text{Pre}_f)$. In particular, $\max(\text{Fix}_f)$ and $\min(\text{Fix}_f)$ exist.

Let now be $M \subseteq \text{Fix}_f$ be an arbitrary subset of $\text{Fix}_f$. Since $(D, \sqsubseteq)$ is a complete lattice, we known that $\sup(M)$ and $\inf(M)$ exists. To prove that $(\text{Fix}_f, \sqsubseteq)$ is also a complete lattice, we have to show that among the fixpoints in $\text{Fix}_f$ there are a least upper and a greatest lower bound of $M$, i.e., $\min(\{ u \in \text{Fix}_f \mid \forall x \in M. x \sqsubseteq u \})$ and $\max(\{ u \in \text{Fix}_f \mid \forall x \in M. u \sqsubseteq x \})$ exist. We only prove the existence of $\inf(\{ u \in \text{Fix}_f \mid \forall x \in M. x \sqsubseteq u \})$, since the remaining proof is symmetric.

To this end, we consider the set of upper bounds $U_M := \{ u \in D \mid \sup(M) \subseteq u \}$ of $M$. By construction of $U_M$, we have $\sup(M) \in U_M$ and (4) $\sup(M) = \min(U_M)$. We now prove that for any upper bound $u \in U_M$, its image $f(u)$ is also an upper bound, i.e., (5) $\forall u \in U_M. f(u) \in U_M$. This holds since for any $u \in U_M$, we have $\forall x \in M. x \sqsubseteq u$ by definition of $U_M$, and thus by monotonicity of $f$ that $\forall x \in M. f(x) = x \sqsubseteq u \sqsubseteq f(u)$ holds. Thus, $f(u)$ is also an upper bound of $M$, thus $f(u) \in U_M$.

It is easily seen that $(U_M, \sqsubseteq)$ is a complete lattice: Let $X \subseteq U_M$ be a nonempty subset of $U_M$, then we know by the completeness of $D$, which is a superset of $U_M$ that it has a supremum $\sup X$ of $X$ in $D$. Since $\sup X$ is an upper bound of $X$ whose elements are themselves upper bounds of $M$, also $\sup X$ is an upper bound of $M$, and therefore $\sup X \in U_M$ follows. By completeness of $D$, we also conclude that $X$ has an infimum $\inf X$ of $X$ in $D$. We have to prove that $\inf X \in U_M$ holds, i.e., that $\inf X$ is an upper bound of $M$. To this end, note first
that since $m_X$ is the infimum of $X$, we have (6) $\forall y \in D. (\forall x \in X. y \sqsubseteq x) \rightarrow y \sqsubseteq m_X$. Second, note that all elements of $M$ are lower bounds of $U_M$ by definition of $U_M$, and therefore we have (7) $\forall m \in M. \forall u \in U_M. m \sqsubseteq u$. Since, $X \subseteq U_M$ holds, we therefore conclude by (7) that also (8) $\forall m \in M. \forall x \in X. m \sqsubseteq x$ holds and by modus ponens with (6) that also $\forall m \in M. m \sqsubseteq m_X$. Hence, we have $m_X \in U_M$, so that every nonempty subset $X \subseteq U_M$ has both an infimum and a supremum in $U_M$.

For this reason, the restriction of $f_{|U_M}$ has type $f_{|U_M} : U_M \rightarrow U_M$, and since $(U_M, \sqsubseteq)$ is a complete lattice, we know by (***) above that $f_{|U_M}$ has a least fixpoint in $U_M$, i.e. $\min(\{u \in U_M \mid f(u) = u\})$ exists. It is now easy to see that $\{u \in U_M \mid f(u) = u\} = \{u \in \text{Fix}_f \mid \forall x \in M. x \sqsubseteq u\}$, so that $\min(\{u \in U_M \mid f(u) = u\}) = \min(\{u \in \text{Fix}_f \mid \forall x \in M. x \sqsubseteq u\})$ exists. $\square$

The existence of the least and greatest fixpoints in the above proof can alternatively be obtained from the fact that $(D, \sqsubseteq)$ and $(D, \sqsubseteq^{-1})$ are complete partial orders, so that Theorem A.17 suffices for this proof.

Hence, similar to complete partial orders, we know that every monotone function has fixpoints, but we have no means for computing them. Clearly, the situation changes again for continuous functions $f$ [153, 251]:

**Theorem A.33 (Fixpoints of Continuous Functions (Tarski/Knaster (II))).**

Let $(D, \sqsubseteq)$ be a complete lattice, $f : D \rightarrow D$ be a continuous function, and let $p, q \in D$ be such that the following conditions hold:

$$(\mu_1) \ p \sqsubseteq f(p) \quad (\nu_1) \ f(q) \sqsubseteq q$$

$$(\mu_2) \ p \sqsubseteq \hat{x} \quad (\nu_2) \ \hat{x} \sqsubseteq q$$

Since $f$ is monotone, the set of its fixpoints is itself a complete lattice so that it has in particular a maximal fixpoint $\hat{x}$ and a minimal fixpoint $\hat{x}$. Moreover, the iteration $p_i$ defined as $p_0 := p$ and $p_{i+1} := f(p_i)$ converges to $\hat{x}$, and the iteration $q_i$ defined as $q_0 := q$ and $q_{i+1} := f(q_i)$ converges to $\hat{x}$.

**Proof.** Since $(D, \sqsubseteq)$ is a complete lattice, it follows by Theorem A.31 that $(D, \sqsubseteq)$ is also a complete partial order. Thus, by Theorem A.20, we already have the proof that the sequence $p_i$ converges to the least fixpoint of $f$. Since by Theorem A.31 $(D, \sqsubseteq^{-1})$ is also a complete partial order, and $f$ is also continuous in that complete partial order, it follows again by Theorem A.20 that the sequence $q_i$ converges to the least fixpoint of $f$ wrt. $\sqsubseteq^{-1}$, i.e., to the greatest fixpoint of $f$ wrt. to $\sqsubseteq$. $\square$

In general, the above fixpoint iterations converge to the least or greatest fixpoint. However, the fixpoint, i.e., the limit $\lim_{i \rightarrow \infty} f^i(p)$ of the above mentioned iterations may never be actually reached. For finite $D$, however, there must be a value $n_0$ such that $f^{n_0}(p) = f^{n_0+1}(p) = \lim_{i \rightarrow \infty} f^i(p)$ holds, hence the iteration will always terminate for finite lattices.

Hence, we have seen that if a lattice is complete, then every monotone function has fixpoints. It is remarkable that also the converse holds [75]:
Theorem A.34 (Least Fixpoint Characterization of Complete Lattices). A lattice \((D, \sqsubseteq)\) is a complete iff every monotone function \(f : D \to D\) has a fixpoint.

A.3 Theory of Streams

In this section, we consider the mathematics of streams that will be required for the definition of the semantics of DPNs. To this end, we consider a set of data values \(D\) that is not further specified. For a real programming language, it will be further partitioned into different data types, which would however make the explanations of this chapter more complicated than necessary. For the set \(D\), we consider the sets of finite sequences \(D^*\) and infinite sequences \(D^\omega\) as well as their union \(\mathcal{D} := D^* \cup D^\omega\).

As process nodes of a DPN may read data from several input streams and may also write output values to several output streams, we will describe their semantics as functions of type \(f : \mathcal{D}^m \to \mathcal{D}^n\). In order to simplify the notation, we abbreviate \(\mathcal{D}^k\) as \(\text{Stream}_k\), so that we will consider functions of type \(f : \text{Stream}_n^D \to \text{Stream}_n^D\). The sets \(\text{Stream}_n^D\) are called the sets of streams over \(D\). Moreover, we say that a stream \(\sigma = (\sigma_1, \ldots, \sigma_n) \in \text{Stream}_n^D\) is finite if all its components \(\sigma_i\) are finite, otherwise \(\sigma\) is an infinite stream.

Since we have to study such functions, we will consider in the following the mathematical properties of the set of streams \(\text{Stream}_n^D\). We first show how \(\text{Stream}_n^D\) can be endowed with a relation \(\sqsubseteq\) that makes \(\text{Stream}_n^D\) a complete partial order. Most research has considered this natural formal foundation of the theory of streams. A second foundation is obtained by introducing a distance \(\Delta\) between streams which makes \(\text{Stream}_n^D\) a metric space. In both cases, we can derive theorems that guarantee the existence of fixpoints which in turn guarantee the determinism of the networks.

A.3.1 Complete Partial Order of Streams

We will first define a partial order \(\sqsubseteq\) on streams \(\text{Stream}_n^D\) that will turn out to be even a complete partial order. To this end, we consider the concatenation of finite sequences, or of a finite sequence with an infinite sequence. The concatenation can also be performed componentwise on tuples of sequences, where the first (left) one has to consist of finite streams only. Based on this concatenation, we can define the following prefix ordering:

Definition A.35 (Prefix Ordering). For \(\sigma_1, \sigma_2 \in \text{Stream}_n^D\), we define the prefix ordering \(\sigma_1 \sqsubseteq \sigma_2 \iff \sigma_1 = \sigma_2 \lor \exists \beta \in \text{Stream}_n^D. \sigma_1 \beta = \sigma_2\).

It is easily seen that \((\text{Stream}_n^D, \sqsubseteq)\) is a partial order. Moreover, note that for two streams \(\sigma_1, \sigma_2 \in (D^\omega)^n\) that only consist of infinite sequences, we have

\[\sigma_1 \sqsubseteq \sigma_2 \iff \sigma_1 = \sigma_2\]

In research papers, the meaning of \(D^\omega\) and \(D^\omega\) is often reversed which is unfortunately inconsistent with the theory of \(\omega\)-automata.
The main result of this subsection is to prove that \((\text{Stream}^n_D, \preceq)\) is even a complete partial order. To this end, we will first prove that \((\text{Stream}^1_D, \preceq)\) is a cpo so that the completeness of \((\text{Stream}^n_D, \preceq)\) follows from Theorem A.22. To this end, the following lemma about directed subsets of \(\text{Stream}^1_D\) is convenient:

**Lemma A.36 (Directed Subsets in \((\text{Stream}^1_D, \preceq)\)).** Every directed subset \(M \subseteq \text{Stream}^1_D\) of \((\text{Stream}^1_D, \preceq)\) is totally ordered by \(\preceq\), and one of the following cases applies:

- \(M\) consists of finitely many finite streams that can be arranged in a chain \(\sigma_0 \preceq \ldots \preceq \sigma_p\), so that \(\sigma_p\) is its maximum.
- \(M\) consists of infinitely many finite streams that can be arranged in an infinite chain \(\sigma_0 \preceq \sigma_1 \preceq \ldots\), so that there is no maximum.
- \(M\) has an infinite stream that is then the only infinite stream and also the maximum of \(M\).

**Proof.** We first prove that \(\preceq\) is a total order on each directed subset \(M \subseteq \text{Stream}^1_D\). To this end, assume \(M\) is a directed subset of \(\text{Stream}^1_D\) and we have \(\sigma_1, \sigma_2 \in M\). We have to prove that either \(\sigma_1 \preceq \sigma_2\) or \(\sigma_2 \preceq \sigma_1\) holds: Since \(M\) is a directed subset, there is a \(\bar{\sigma} \in M\) such that \(\sigma_1 \preceq \bar{\sigma}\) and \(\sigma_2 \preceq \bar{\sigma}\). By the definition of \(\preceq\), there are now two cases: First, if \(\sigma_1 = \bar{\sigma}\) holds, then it follows directly that \(\sigma_2 \preceq \bar{\sigma} = \sigma_1\) holds. Otherwise, there exists \(\beta_1 \in \text{Stream}^1_D\) with \(\sigma_1 \beta_1 = \bar{\sigma}\), and therefore we have \((\ast)\) \(\sigma_2 \preceq \sigma_1 \beta_1\). A further case distinction on \((\ast)\) reveals that either \(\sigma_2 = \sigma_1 \beta_1\) holds, so that directly \(\sigma_1 \preceq \sigma_2\) follows, or that there is a \(\beta_2 \in \text{Stream}^1_D\) with \(\sigma_1 \beta_1 = \sigma_2 \beta_2\). In that case, both \(\sigma_1\) and \(\sigma_2\) are finite prefixes of the same potentially infinite sequence. Hence, one of the two has a length less than or equal to the length of the other one, and is therefore a prefix of the other one, so that \(\sigma_1 \preceq \sigma_2\) or \(\sigma_2 \preceq \sigma_1\) holds.

Next, consider two cases: In the first case, \(M\) contains infinite streams. Assume first that there were infinite streams \(\sigma_1, \sigma_2 \in M \cap D^\omega\). Since \(M\) is directed, it follows that there is a \(\bar{\sigma} \in M\) with \(\sigma_1 \preceq \bar{\sigma}\) and \(\sigma_2 \preceq \bar{\sigma}\). By definition of \(\preceq\), it follows that \(\bar{\sigma}\) must also be an infinite stream, and that \(\sigma_1 = \bar{\sigma} = \sigma_2\) holds. Thus, if \(M\) should contain an infinite stream \(\sigma_1\), then that stream \(\sigma_1\) is the only stream in \(M\) that is infinite. Moreover, that infinite stream \(\sigma_1\) must be the maximum of \(M\) since for it and any other member of \(M\), there must be an upper bound in \(M\) which can only be \(\sigma_1\) itself.

In the second case, \(M\) contains only finite streams. Since the elements of \(M\) are totally ordered and finite, it follows that the elements of \(M\) can be arranged in a sequence \(\sigma_0 \preceq \sigma_1 \preceq \ldots\). If \(M\) should have only finitely many elements, then this sequence is finite, i.e. \(\sigma_0 \preceq \ldots \preceq \sigma_p\), and ends with the maximal element \(\sigma_p\) of \(M\). Otherwise, the sequence is infinite, so that each \(\sigma_i\) is a prefix of its successor \(\sigma_{i+1}\). In this case, there is no maximal element. \(\Box\)
The above lemma is the key to prove the completeness of \((\text{Stream}^1_D, \preceq)\) below. It can however not be generalized to \(\text{Stream}^n_D\) with \(n > 1\): As an example, note that we neither have \((aa, b) \preceq (a, bb)\) nor \((a, bb) \preceq (aa, b)\), so that the directed set \(\{(a, bb), (aa, b), (aa, bb)\}\) is not totally ordered. However, \((aa, bb)\) is clearly the maximum of \(M\) that must exist due to Lemma A.7.

Using the above lemma, it is now straightforward to prove the completeness of \((\text{Stream}^1_D, \preceq)\):

**Lemma A.37 (Complete Partial Order of Streams (\(\text{Stream}^1_D, \preceq))\).**

The partial order \((\text{Stream}^1_D, \preceq)\) of streams is complete.

*Proof.* To prove that \((\text{Stream}^1_D, \preceq)\) is a complete partial order, we have to prove that every directed subset \(M \subseteq \text{Stream}^1_D\) has a supremum in \(\text{Stream}^1_D\). By the previous lemma, this is true for each directed subset \(M\) that has an infinite stream or that consist of only finitely many finite streams (since in these cases \(M\) has even a maximal element). In the remaining case, \(M\) consists of infinitely many finite streams that can be arranged in a strictly increasing sequence \(\sigma_0 \preceq \sigma_1 \preceq \ldots\). We now determine the supremum of \(M\) as an infinite stream \(\hat{\sigma}\) that we define as follows: \(\hat{\sigma}^{(i)} := \sigma_i^{(i)}, \) i.e., the \(i\)-th element of \(\hat{\sigma}\) is the \(i\)-th element of \(\sigma_i\). Note that since the sequence \(\sigma_0 \preceq \sigma_1 \preceq \ldots\) is strictly increasing, it follows that the length of \(\sigma_i\) must be at least \(i\). It is easily seen that each finite sequence \(\sigma_i\) is a prefix of \(\hat{\sigma}\), so that \(\hat{\sigma}\) is an upper bound of \(M\). It is also easily seen that it is the least upper bound.

As we now know that \((\text{Stream}^1_D, \preceq)\) is a complete partial order, and we even know how to compute suprema of directed subsets, it is now a small step to prove the final result of this section:

**Theorem A.38 (Complete Partial Order of Streams (\(\text{Stream}^n_D, \preceq))\).**

The partial order \((\text{Stream}^n_D, \preceq)\) of streams is complete.

*Proof.* It is easily seen that \((\text{Stream}^n_D, \preceq)\) is the \(n\)-fold product of \(\text{Stream}^1_D\), so that its completeness follows from Lemma A.37 and Theorem A.22.

Since \((\text{Stream}^n_D, \preceq)\) is a complete partial order, let us consider what monotonicity and continuity of functions \(f : \text{Stream}^m_D \rightarrow \text{Stream}^n_D\) mean. To this end, imagine that \(f\) is implemented by a DPN, so that \(\beta := f(\alpha)\) means that \(\alpha\) is read from input buffers and \(\beta\) is written to the output buffers. **Monotonicity means then that the more input values are consumed, the more output values were produced. Continuity additionally means that the function’s values for infinite input streams are smoothly obtained from convergence of its finite prefixes, i.e., for an infinite chain \(\sigma_0 \preceq \sigma_1 \preceq \ldots\) with supremum \(\hat{\sigma} := \lim_{i \to \infty} \sigma_i\), we must have \(f(\hat{\sigma}) = f(\lim_{i \to \infty} \sigma_i) = \lim_{i \to \infty} f(\sigma_i)\).**

In order to derive more about monotone and continuous functions on streams, we start with the following simple lemma:

**Lemma A.39 (Monotone Functions).** A function \(f : \text{Stream}^m_D \rightarrow \text{Stream}^n_D\) is monotone iff for all \(\sigma_1 \in (D^*)^m, \sigma_2 \in \text{Stream}^m_D\), we have \(f(\sigma_1) \preceq f(\sigma_1 \sigma_2)\).
Proof. We split the proof into two implications:

‘⇒’: Assume \( f \) is monotone. Since \( \sigma_1 \subseteq \sigma_1 \sigma_2 \) holds, we have \( f(\sigma_1) \subseteq f(\sigma_1 \sigma_2) \).

‘⇐’: Assume we have (1) \( f(\sigma_1) \subseteq f(\sigma_1 \sigma_2) \) for all \( \sigma_1 \in (D^*)^m \), \( \sigma_2 \in \text{Stream}_D^n \).

For arbitrary streams \( \alpha, \beta \) with \( \alpha \subseteq \beta \), we have to prove that \( f(\alpha) \subseteq f(\beta) \) holds. By definition of \( \subseteq \), we either have \( \alpha = \beta \) or \( \alpha \gamma = \beta \) for some stream \( \gamma \). In the first case, we clearly have \( f(\alpha) = f(\beta) \), and therefore \( f(\alpha) \subseteq f(\beta) \). In the second case, our proof goal is equivalent to \( f(\alpha) \subseteq f(\alpha \gamma) \) which holds by our assumption (1).

\[ \square \]

Note that \( f(\sigma_1) \subseteq f(\sigma_1 \sigma_2) \) means by definition of \( \subseteq \) that either \( f(\sigma_1 \sigma_2) = f(\sigma_1) \) or \( f(\sigma_1 \sigma_2) = f(\sigma_1) \cdot \beta \) holds. Hence, every monotone function may only concatenate a sequence \( \beta \) to the right end of an already computed image \( f(\sigma_1) \) of a prefix \( \sigma_1 \). Conversely, every function that reads an input sequence step by step and computes thereby the corresponding output sequence also step by step is therefore monotone. Hence, monotonicity of the functions implemented by the nodes of DPN is immediately given.

There is also a need to extrapolate the infinite behavior of a stream function by its finite behavior, which is of course possible iff the function is continuous. If we are only given a monotone function \( f : (D^*)^m \rightarrow (D^*)^n \), then we can extend it to a uniquely determined continuous function \( g : \text{Stream}_D^m \rightarrow \text{Stream}_D^n \) as stated in the following theorem:

**Theorem A.40 (Continuous Extension of Monotone Functions).** Given a monotone function \( f : (D^*)^m \rightarrow (D^*)^n \), there is a uniquely determined continuous function \( g : \text{Stream}_D^m \rightarrow \text{Stream}_D^n \) such that \( f(\sigma) = g(\sigma) \) for every \( \sigma \in (D^*)^m \).

**Proof.** For any arbitrary directed subset \( M \subseteq \text{Stream}_D^m \), we consider the set \( \text{Prefix}(M) := \{ \sigma \in (D^*)^m \mid \sigma \subseteq \text{sup}(M) \} \). We then define \( g \) as follows

\[
(1) \quad g(\sigma) := \begin{cases} f(\sigma) & : \text{if } \sigma \text{ is finite} \\ \text{sup}(f(\text{Prefix}(\{\sigma\}))) & : \text{if } \sigma \text{ is infinite} \end{cases}
\]

It is easily seen that (2) \( g(\sigma) = \text{sup}(f(\text{Prefix}(\{\sigma\}))) \) holds also for finite \( \sigma \):
In that case, we have \( \sigma \in \text{Prefix}(\{\sigma\}) \) and even \( \sigma = \text{max}(\text{Prefix}(\{\sigma\})) \), so that by monotonicity of \( f \) and the finiteness of \( \text{Prefix}(\{\sigma\}) \), we conclude that \( f(\sigma) = \text{max}(f(\text{Prefix}(\{\sigma\}))) = \text{sup}(f(\text{Prefix}(\{\sigma\}))) = g(\sigma) \).

Note that we have \( \text{sup}(M) = \text{sup}(\text{Prefix}(M)) \), but we will not need this in the following. Instead, we need the following facts:

- (3) \( \text{Prefix}(M) = \text{Prefix}(\{\text{sup}(M)\}) \)
- (4) \( \text{sup}(\bigcup_{i \in I} \text{sup}(M_i)) = \text{sup}(\bigcup_{i \in I} M_i) \) for arbitrary sets \( M_i \) over an index set \( I \) (provided that the suprema exist)

(3) is easily seen; just note that \( \text{sup}(\{\text{sup}(M)\}) = \text{sup}(M) \) holds, and consider the definition of \( \text{Prefix} \). (4) is left to the reader.
To prove that \( g \) is continuous, consider a directed subset \( M \subseteq \text{Stream}^m_D \), so that we have to prove that \( g(\sup(M)) = \sup(g(M)) \) holds. This is seen follows:

\[
\begin{align*}
\sup(g(M)) &= \sup\{g(\sigma) \mid \sigma \in M\} \\
&= \sup(\{\sup(f(\text{Prefix}(\{\sigma\}))) \mid \sigma \in M\}) \\
&= \sup(\{f(\text{Prefix}(\{\sigma\})) \mid \sigma \in M\}) \\
&= \sup(\text{Prefix}(M)) \\
&= \sup(\{ \sup(\text{Prefix}(\{\sigma\})) \mid \sigma \in M\}) \\
&= g(\sup(M)) \quad (2)
\end{align*}
\]

The above result is remarkable, since it does not hold for general cpos. It has also the interesting following consequence:

\textbf{Corollary A.41.} A monotone function \( f : \text{Stream}^m_D \rightarrow \text{Stream}^n_D \) is continuous iff for all \( \sigma \in \text{Stream}^m_D \), we have \( f(\sigma) = \sup(f(\text{Prefix}(\{\sigma\}))) \).

For this reason, monotone functions \( f : \text{Stream}^m_D \rightarrow \text{Stream}^n_D \) that are not continuous are somewhat artificial, and can be easily made continuous according to the above theorem. For example, a monotone function that is not continuous is \( f : \text{Stream}^1_{[1,0]} \rightarrow \text{Stream}^1_{[1,0]} \) where:

\[
f(\sigma) := \begin{cases} 
[0] & \text{if } \sigma \text{ is finite} \\
[0, 1] & \text{if } \sigma \text{ is infinite}
\end{cases}
\]

Clearly, \( f \) is monotone, but since for any infinite \( \sigma \), we have \( f(\sigma) = [0, 1] \neq [0] = \sup(f(\text{Prefix}(\{\sigma\}))) \), \( f \) is not continuous. It can be easily made continuous by defining \( f(\sigma) = [0] \) for every \( \sigma \).

In particular, we will obtain from the operational semantics of dataflow process networks a monotone function \( f : (\mathcal{D}^*)^m \rightarrow (\mathcal{D}^*)^n \) that will then be extended to \( f : \text{Stream}^m_D \rightarrow \text{Stream}^n_D \) according to Theorem A.40.

\textbf{A.3.2 Metric Space of Streams}

In the previous section, we have studied the complete partial order of streams \((\text{Stream}^m_D, \preceq)\) with the prefix ordering relation \( \preceq \). This allows us to represent infinite streams as limit of a convergent sequence of finite sequences. In this subsection, we proceed by another view on streams that is given by a definition of a distance of two streams. This makes the set of streams a metric space so that additional knowledge can be applied.

\textbf{Definition A.42 (Distance and Neighborhood of Streams).} For streams \( \sigma_1, \sigma_2 \in \text{Stream}^1_D \), we define their distance as a non-negative real number as follows:
\(\delta(\sigma_1, \sigma_2) := \begin{cases} 
0 & : \text{if } \sigma_1 = \sigma_2 \\
\frac{1}{\sigma_1 + 1} & : \text{if } \sigma_1 \text{ is a finite prefix of } \sigma_2 \neq \sigma_1 \\
\frac{1}{i + 1} & : \text{if } i \text{ is the first position where } \sigma_1^{(i)} \neq \sigma_2^{(i)}
\end{cases}\)

For streams \((\sigma_1, \ldots, \sigma_n), (\sigma'_1, \ldots, \sigma'_n) \in (\text{Stream}_n^n)\), we define their distance as a non-negative real number as follows:

\[\Delta((\sigma_1, \ldots, \sigma_n), (\sigma'_1, \ldots, \sigma'_n)) := \max\{\delta(\sigma_i, \sigma'_i) \mid i = 1, \ldots, n\}\]

For every \(\varepsilon > 0\) and \(\sigma \in \text{Stream}^n_\mathbb{R}\), we define \(\text{Sphere}_{\varepsilon}(\sigma) := \{\sigma' \in \text{Stream}^n_\mathbb{R} \mid \Delta(\sigma, \sigma') \leq \varepsilon\}\).

It can be easily seen that we have \(0 \leq \Delta(\sigma_1, \sigma_2) \leq 1\), where \(\Delta(\sigma_1, \sigma_2) = 0\) holds iff \(\sigma_1 = \sigma_2\) holds, and \(\Delta(\sigma_1, \sigma_2) = 1\) holds iff one of the streams is empty and the other is nonempty or when both differ in the first element. The definition of the distance \(\Delta(\sigma, \sigma')\) of streams makes \(\text{Stream}^n_\mathbb{R}\) a metric space so that we can make use of results of topology:

**Theorem A.43 (Metric Space of Streams).** \((\text{Stream}^n_\mathbb{R}, \Delta(\cdot, \cdot))\) is a metric space, since we have:

- \(\Delta(\sigma_1, \sigma_2) \geq 0\) (non-negativity)
- \(\Delta(\sigma_1, \sigma_2) = 0\) iff \(\sigma_1 = \sigma_2\) (identity of indiscernibles)
- \(\Delta(\sigma_1, \sigma_2) = \Delta(\sigma_2, \sigma_1)\) (symmetry)
- \(\Delta(\sigma_1, \sigma_2) \leq \Delta(\sigma_1, \sigma_3) + \Delta(\sigma_3, \sigma_4)\) (triangle inequality).

The definition of \(\Delta(\sigma, \sigma')\) in terms of \(\delta(\sigma_i, \sigma'_i)\) is somehow arbitrary in that many different distance measures could be used instead. However, the following different measures are all equivalent for considering spheres \(\text{Sphere}_{\varepsilon}(\sigma)\) and therefore they all define somehow equivalent neighborhoods:

**Lemma A.44 (Jensen’s Inequality).** Given positive numbers \(x_1, \ldots, x_n\), and numbers \(0 < p < q\), we have

\[q \left( \sum_{i=1}^{n} x_i^q \right)^{\frac{1}{q}} \leq p \left( \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}} \text{ and } \lim_{p \to \infty} p \left( \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}} = \max\{x_1, \ldots, x_n\}\]

The proof of the above lemma is not difficult and can be found in many textbooks. It is well-known that we could make use of the above formulas by defining a metric \(\Delta_p(\sigma_1, \sigma_2) := \sqrt[p]{\sum_{i=1}^{n} \delta(\sigma_i, \sigma'_i)^p}\) to define a metric on \(\text{Stream}^n_\mathbb{R}\). For discussion convergence, as done below, it does not make a difference which of the metrics \(\Delta_p\) or \(\Delta_\infty\) is chosen, since one can further prove that they all behave equivalent concerning convergence. One can even prove that there are constants \(c_1\) and \(c_2\) so that \(c_1 \Delta_p(\sigma_1, \sigma_2) \leq \Delta_\infty(\sigma_1, \sigma_2) \leq c_2 \Delta_p(\sigma_1, \sigma_2)\) for all \(\sigma_1, \sigma_2\). Moreover, convergence as defined below is equivalent to componentwise convergence, i.e., for \(\sigma_i = (\sigma_{i,1}, \ldots, \sigma_{i,n})\), we have \(\lim_{i \to \infty} \sigma_i = (\lim_{i \to \infty} \sigma_{i,1}, \ldots, \lim_{i \to \infty} \sigma_{i,n})\).
A sequence $σ_0, σ_1, \ldots$ of streams converges to the limit $σ ∈ \text{Stream}_D^n$ if for every $ε > 0$ there exists an $n ∈ \mathbb{N}$ such that for all $i ≥ n$, we have $Δ(σ_i, σ) < ε$. The limit of a convergent sequence $σ_i$ is denoted as $\lim_{i→∞} σ_i$. One can easily prove that $\lim_{i→∞} σ_i$ is uniquely determined for a convergent sequence.

A sequence $σ_0, σ_1, \ldots$ is a Cauchy sequence if for every $ε > 0$, there is a $n_0 ∈ \mathbb{N}$ such that for all $m, n > n_0$, we have $Δ(σ_m, σ_n) < ε$. It is easily seen that every convergent sequence is also a Cauchy sequence, the converse is however not true: In the metric space $\{x ∈ \mathbb{R} \mid 0 < x < 1\}$ with distance $Δ(x_1, x_2) := |x_2 - x_1|$, the sequence $x_i := \frac{1}{1+i}$ is a Cauchy sequence, but it does not converge in $\{x ∈ \mathbb{R} \mid 0 < x < 1\}$. A metric space where each Cauchy sequence converges is called a complete metric space, and fortunately, the streams form a complete metric space:

**Theorem A.45 (Complete Metric Space of Streams).** $(\text{Stream}_D^n, Δ(\cdot, \cdot))$ is a complete metric space, i.e., if for every sequence $σ_0, σ_1, \ldots$ of streams we have $\lim_{n,m→∞} Δ(σ_m, σ_n) = 0$, then there is a stream $σ ∈ \text{Stream}_D^n$ with $σ = \lim_{n→∞} σ_n$.

A particular useful theorem on complete metric spaces is Banach’s fixpoint theorem:

**Theorem A.46 (Banach’s Fixpoint Theorem).** Assume that for function $f : \text{Stream}_D^n → \text{Stream}_D^n$ on the metric space $(\text{Stream}_D^n, Δ(\cdot, \cdot))$ there is a real number $q < 1$ such that for all $x, y ∈ D$, we have $Δ(f(x), f(y)) < q ⋅ Δ(x, y)$. Then, $f$ has a unique fixpoint $x_f ∈ D$ that is the limit of the sequence $x_{i+1} = f(x_i)$ starting with an arbitrary element $x_0$. We moreover have the following estimation on the speed of the convergence:

$$Δ(x_f, x_i) ≤ \frac{q^i}{1-q} ⋅ Δ(x_1, x_0)$$

**Proof.** The proof is not difficult: We can easily show by induction on $n$ that

(1) $Δ(x_{n+1}, x_n) ≤ q^n ⋅ Δ(x_1, x_0)$ holds: The induction base is trivial, and $Δ(x_{n+2}, x_{n+1}) = Δ(f(x_{n+1}), f(x_n)) < q ⋅ Δ(x_{n+1}, x_n) ≤ q ⋅ q^n ⋅ Δ(x_1, x_0) = q^{n+1} ⋅ Δ(x_1, x_0)$.

Based on (1), we can prove by another induction on $k$ that the following inequality (2) $∀k, ∀n. Δ(x_{n+k}, x_n) ≤ q^n \frac{1-q^k}{1-q} ⋅ Δ(x_1, x_0)$ is also true: The induction base is again trivial, and the induction step is seen as follows:

$$Δ(x_{n+k+1}, x_n) ≤ Δ(x_{n+k+1}, x_{n+k}) + Δ(x_{n+k}, x_n) \quad \text{(by triangle inequality)}$$

$$(1) \leq q^{n+k} ⋅ Δ(x_1, x_0) + Δ(x_{n+k}, x_n)$$

$$(IH) \leq q^{n+k} ⋅ Δ(x_1, x_0) + q^n \frac{1-q^k}{1-q} ⋅ Δ(x_1, x_0)$$

$$= \left( q^{n+k} + q^n \frac{1-q^k}{1-q} \right) ⋅ Δ(x_1, x_0)$$

$$= q^n \frac{1-q^{k+1}}{1-q} ⋅ Δ(x_1, x_0)$$

$$≤ q^n \frac{1-q^k}{1-q} ⋅ Δ(x_1, x_0)$$
Now, consider an arbitrary $\varepsilon > 0$. To prove that $x_i$ is a Cauchy sequence, we show that there is a $n_0 \in \mathbb{N}$ such that for all $n > n_0$ and $k > 1$, we have $\Delta(x_{n+k}, x_n) < \varepsilon$. By (2), this is proved if we can show that there is a $n_0 \in \mathbb{N}$ such that for all $n > n_0$ and $k > 1$, we have $q^n \frac{1-q^k}{1-q} \cdot \Delta(x_1, x_0) < \varepsilon$, i.e., $q^n < \frac{\varepsilon}{\Delta(x_1, x_0)} \frac{1-q}{1-q^k}$. This is true since $\lim_{n \to \infty} q^n = 0$ (since $0 < q < 1$).

Thus, $x_i$ is a Cauchy sequence, and therefore $x_f := \lim_{n \to \infty} x_i$ exists. It remains to prove that $x_f$ is a fixpoint and that it is the only one. To see that $x_f$ is a fixpoint, note

$$0 \leq \Delta(x_{n+1}, x_f) = \Delta(f(x_n), f(x_f)) < q \cdot \Delta(x_n, x_f)$$

Since $\lim_{n \to \infty} \Delta(x_n, x_f) = 0$ holds, it follows by the above sandwich inequalities that $\lim_{n \to \infty} \Delta(x_{n+1}, f(x_f)) = 0$. Hence, $\lim_{n \to \infty} x_{n+1} = \lim_{n \to \infty} x_n = f(x_f)$. Since the limit of a convergent sequence is however unique, we thus have $f(x_f) = x_f$, so that $x_f$ is a fixpoint of $f$.

Finally, assume there would be another fixpoint $y \neq x_f$. Then, we would have

$$\Delta(x_f, y) = \Delta(f(x_f), f(y)) < q \cdot \Delta(x_f, y)$$

Since $y \neq x_f$ holds, we have $\Delta(x_f, y) > 0$, so that $1 < q$ would follows. By assumption, we have however $q < 1$, so there is no other fixpoint $y$. \qed

An application of Banach’s fixpoint theorem can be obtained as soon as we find functions that satisfy the property $\Delta(f(x), f(y)) < q \cdot \Delta(x, y)$ for some $q < 1$. It is not difficult to prove that this property holds for contractive functions that are defined as follows:

**Definition A.47 (Causal and Contractive Functions).**

A function $f : \text{Stream}^0_D \to \text{Stream}^D_D$ is *causal* if for all $\sigma_1, \sigma_2 \in \text{Stream}^D_D$ and all $n \in \mathbb{N}$, $\Delta(\sigma_1, \sigma_2) \leq \frac{1}{n+1}$ implies $\Delta(f(\sigma_1), f(\sigma_2)) \leq \frac{1}{n+1}$.

The function $f$ is *contractive* if for all $\sigma_1, \sigma_2 \in \text{Stream}^n_D$ and all $n \in \mathbb{N}$, $\Delta(\sigma_1, \sigma_2) \leq \frac{1}{n+1}$ implies $\Delta(f(\sigma_1), f(\sigma_2)) \leq \frac{1}{n+2}$.

Intuitively, $f$ is causal if the first $n$ elements of $f(\sigma)$ are determined by the first $n$ elements of $\sigma$, and $f$ is contractive if the $n$ elements of $f(\sigma)$ are even determined by the first $n-1$ elements of $\sigma$. To see this, note that if $\sigma_1$ and $\sigma_2$ share a common prefix of length $n$, then and only then, we have $\Delta(\sigma_1, \sigma_2) \leq \frac{1}{n+1}$.

**Lemma A.48 (Contractive Functions).** If $f : \text{Stream}^n_D \to \text{Stream}^D_D$ is contractive, then there is a $q < 1$ with $\Delta(f(x), f(y)) < q \cdot \Delta(x, y)$.

**Proof.** Assume by contradiction that $f$ is contractive, but that $\Delta(f(\sigma_1), f(\sigma_2)) \geq \Delta(\sigma_1, \sigma_2)$. If $\sigma_1$ and $\sigma_2$ share a common prefix of length $\geq n$, then we have $\Delta(\sigma_1, \sigma_2) \leq \frac{1}{n+1}$, and since $f$ is contractive, it follows $\Delta(f(\sigma_1), f(\sigma_2)) \leq \frac{1}{n+2} < \frac{1}{n+1}$. However, this implies the following contradiction
\[
\frac{1}{n+1} \leq \Delta(\sigma_1, \sigma_2) \leq \Delta(f(\sigma_1), f(\sigma_2)) \leq \frac{1}{n+2} < \frac{1}{n+1}
\]

We therefore obtain by Banach’s fixpoint theorem the following result:

**Corollary A.49 (Contractive Functions).** If \( f : \text{Stream}^n_D \rightarrow \text{Stream}^n_D \) is contractive, then it has a unique fixpoint.

Note that Banach’s fixpoint theorem cannot be applied to causal functions as well. We therefore need stronger results to assure the fixpoint semantics of causal functions, since we need an additional fixpoint computation for the reaction in each step of the computation.
As it is the case for every programming language with numeric data types, the semantics of the Quartz language depends on the semantics of the provided arithmetic operators for signed/unsigned bounded/unbounded integers. Except for division and modulo operation, these operations are standard and are therefore presented in this appendix to keep them separated from the more specific issues concerning synchronous programming. Good introductory texts on computer arithmetic are for example [110, 129, 146, 148, 202]. For complete implementations of multiprecision arithmetic see e.g. [61, 101, 240] or many other resources like the GNU multiprecision library.

The human mankind has spent hundreds of centuries to investigate different kinds of numbers, number representations and efficient algorithms to perform arithmetic operations. Some ancient civilizations already had an incredibly large knowledge of arithmetics. In particular, the Maya and the Babylonians developed complex arithmetics for their research in astronomy. The Indian nations invented the zero, a number that does not naturally occur in practice, but that frequently occurs in algorithms. However, only in the last two centuries, people tried to think about what numbers really are. To this end, they developed abstract axiom systems to specify what properties must hold to describe natural and other numbers. It was the breakthrough of Peano and Dedekind [78, 201] to develop small axiom systems from which all relevant theorems of arithmetic can be derived.

In this chapter, we follow in Section B.1 Peano’s axiomatic approach, and postulate his axioms to define what we mean with natural numbers. Based on these axioms, we give definitions of the arithmetic operations we wish to implement. Although these definitions are already in an executable form, they rely on a unary number system that leads to very inefficient algorithms for arithmetics. For this reason, we develop a theory of radix-\(B\) numbers in Section B.2. In particular, we will show how the formerly defined arithmetic operations can be directly implemented on radix-\(B\) numbers, i.e., without decoding and encoding in the radix-\(B\) number system. In Sections B.3 and B.4, we extend these results to signed numbers, so that we can also deal with...
negative numbers. To this end, we generalize the Peano axioms to define the integers and develop then in analogy to the radix-$B$ numbers a theory of $B$-complement numbers.

The algorithms presented in these sections are the core of computer arithmetic and are used here to define the semantics of the Quartz language. In contrast to Esterel, the representation of signed and unsigned integers is fixed in Quartz: In Quartz, binary representation and two's-complement representations are used for the types $\text{nat}^{<n>}$, $\text{int}^{<n>}$, $\text{nat}$, and $\text{int}$ to define the semantics of the arithmetic operators. Moreover, the hardware synthesis of compilers for Quartz should follow the algorithms with $B = 2$, while the software synthesis may use higher radices depending on the microprocessor's bitwidth.

B.1 Natural Numbers

As already mentioned in the introductory text, everybody intuitively believes to know what numbers are and how to deal with them. However, for a formal treatment that is necessary to reason about the correctness of arithmetic algorithms, we need a precise definition of numbers. Such a precise definition has been given by Giuseppe Peano (1858–1932) \[78, 201\] which are the famous Peano axioms\(^1\) to define the natural numbers:

**Definition B.1 (Peano's Definition of Natural Numbers).** The set of natural numbers $\mathbb{N}$ includes a special element 0 (called zero) and provides a successor operation $\text{succ}: \mathbb{N} \rightarrow \mathbb{N}$ so that the following axioms are valid:

\begin{align*}
\text{Peano1:} & \forall n \in \mathbb{N}. 0 \neq \text{succ}(n) \\
\text{Peano2:} & \forall m, n \in \mathbb{N}. \text{succ}(m) = \text{succ}(n) \rightarrow m = n \\
\text{Peano3:} & \forall P. P(0) \land (\forall n \in \mathbb{N}. P(n) \rightarrow P(\text{succ}(n))) \rightarrow \forall n \in \mathbb{N}. P(n)
\end{align*}

In the following, we only assume these axioms and reduce all other results on arithmetics to these axioms. The first axiom simply states that 0 is not the

\(1\) Historically, Peano formulated the following nine axioms:

1. \forall n \in \mathbb{N}. n = n
2. \forall m, n \in \mathbb{N}. m = n \leftrightarrow n = m
3. \forall m, n, p \in \mathbb{N}. m = n \land n = p \rightarrow m = p
4. \forall m, n. m = n \land m \in \mathbb{N} \rightarrow n \in \mathbb{N}
5. 1 \in \mathbb{N}
6. \forall m \in \mathbb{N}. \text{succ}(m) \in \mathbb{N}
7. \forall m \in \mathbb{N}. \neg\text{succ}(m) = 1
8. \forall m, n \in \mathbb{N}. \text{succ}(m) = \text{succ}(n) \leftrightarrow m = n
9. \forall P. P(1) \land (\forall n \in \mathbb{N}. P(n) \rightarrow P(\text{succ}(n))) \rightarrow \forall n \in \mathbb{N}. P(n)

Hence, Peano excluded 0 from $\mathbb{N}$, which is nowadays unusual. Moreover, the first three of the nine axioms define equality, which is what we define in a more general setting and therefore exclude it from the theory of natural numbers.
successor of a natural number, and the second axiom states that the successor function is injective. The most important axiom is the third axiom that is often called the induction axiom. According to this induction principle, we can prove a property \( P \) for all natural numbers \( n \) in that we show that \( P(0) \) holds and that \( P(\text{succ}(n)) \) follows from \( P(n) \) for any \( n \in \mathbb{N} \). An immediate consequence of the induction axiom is that every natural number can be written as an expression of the form \( \text{succ}^n(0) \), i.e., \( n \) function applications of the successor operation to the zero element. This important result is due to Dedekind [78] and proves that the Peano axioms have up to isomorphisms only one model, namely the natural numbers \( \mathbb{N} \). Therefore, we can use these axioms to define the set of natural numbers.

Moreover, since every natural number can be written as \( \text{succ}^n(0) \), we can already use the expressions \( \text{succ}^n(0) \) to represent the natural numbers. This is essentially a unary number system, since the number \( n \) is represented with \( n \) items, in our case, applications of the succ operation. Unary number systems were probably the first number representations since counting things with fingers (which is an attempt to establish an isomorphism between some fingers and other items) is based on a unary number representation just like counting with the well known \( \text{III} \text{II} = 7 \) notation. In the remainder of this chapter, we will work with this unary number system to define arithmetic operations and to prove some basic results about natural numbers.

The induction axiom is a second order formula which makes the theory of natural numbers a second order theory. This has some consequences on its decidability that have been formally proved by Gödel in [115, 116]. Sometimes, people instantiate the induction axiom with all formulas \( P \), so that a first-order axiom system with infinitely many axioms is obtained. However, this is still not equivalent to the second-order axiom system given by Peano.

The induction axiom is also the basis for inductive definitions: According to the induction axiom, a function \( f \) is uniquely defined if we uniquely define its value \( f(0) \) for zero and the value of successor values \( f(\text{succ}(n)) \), where we may use the value of the predecessor \( f(n) \) to define \( f(\text{succ}(n)) \). Hence, every constant \( c_0 \) and every function \( g \) uniquely specify a function \( f_{c,g} \) with \( f_{c,g}(0) := c \) and \( f_{c,g}(\text{succ}(n)) := g(f(n), n) \). Using these primitive recursive definitions, we next define some arithmetic operations:

Using the induction axiom and the corresponding primitive recursive definition principle, we are now able to define the binary arithmetic operators of Figure B.1. Although these definitions do not strictly follow the primitive recursion principle, they are very close to it. For example, the addition operation can be obtained by a function \( f_m \) that we define for a particular number \( m \in \mathbb{N} \) as follows \( f_m(0) := m \) and \( f_m(\text{succ}(n)) := \text{succ}(m + n) \).

Note that the subtraction \( m - n \) yields 0 iff \( m \leq n \) holds, since we do not yet consider negative numbers. The modulo operation simply maps the sequence of natural numbers to the periodic sequence \( 0, \text{succ}^1(0), \text{succ}^2(0), \ldots, \text{succ}^{m-1}(0), 0, \ldots \). Based on this periodic sequence, we increment the
comparison
- $0 \leq m : \Leftrightarrow$ true
- $\text{succ}(n) \leq 0 : \Leftrightarrow$ false
- $\text{succ}(n) \leq \text{succ}(m) : \Leftrightarrow n \leq m$

other comparisons
- $n < m : \Leftrightarrow -(m \leq n)$
- $n > m : \Leftrightarrow m < n$
- $n \geq m : \Leftrightarrow m \leq n$

addition
- $m + 0 = m$
- $m + \text{succ}(n) = \text{succ}(m + n)$

subtraction
- $0 - m = 0$
- $\text{succ}(n) - m = \begin{cases} 0 & : \text{if } n < m \\ \text{succ}(n - m) & : \text{otherwise} \end{cases}$

multiplication
- $m \cdot 0 = 0$
- $m \cdot \text{succ}(n) = m + (m \cdot n)$

modulo operation
- $(0 \mod m) = 0$
- $(\text{succ}(n) \mod m) = \begin{cases} 0 & : \text{if } \text{succ}(n \mod m) = m \\ \text{succ}(n \mod m) & : \text{otherwise} \end{cases}$

division
- $(0 \div m) = 0$
- $(\text{succ}(n) \div m) = \begin{cases} \text{succ}(n \div m) & : \text{if } \text{succ}(n \mod m) = m \\ n \div m & : \text{otherwise} \end{cases}$

distance
- $\delta(m, n) := \begin{cases} m - n & : \text{if } n \leq m \\ n - m & : \text{otherwise} \end{cases}$

Fig. B.1. Primitive Recursive Definitions of Arithmetic Operators

quotient in the division operation each time a period of the modulo sequence has been seen.

For our defined arithmetic operations, we can now prove step-by-step a couple of properties, in particular, those shown in Figure B.2. All of these laws can be easily proved, and the reader is advised to create some of these proofs. By these properties, we can see that the natural numbers fulfill almost the algebraic properties of a ring (the existence of additive inverse elements is missing). Nevertheless, the following fundamental division theorem of Euclidean rings holds, which is of outstanding importance for the entire computer arithmetics since it allows us not only to characterize division and modulo operations, but also to establish radix number systems.

**Theorem B.2 (Division Theorem).** For all numbers $m, n \in \mathbb{N}$ with $n \neq 0$, there are uniquely determined numbers $q$ and $r$ such that

1. $m = q \cdot n + r$
B.1 Natural Numbers

- **commutativity**
  - \( \forall a, b. a + b = b + a \)
  - \( \forall a, b. a \cdot b = b \cdot a \)
- **associativity**
  - \( \forall a, b, c. (a + b) + c = a + (b + c) \)
  - \( \forall a, b, c. (a \cdot b) \cdot c = a \cdot (b \cdot c) \)
- **distributivity**
  - \( \forall a, b, c. (a + b) \cdot c = (a \cdot c) + (b \cdot c) \)
  - \( \forall a, b, c. a \cdot (b + c) = (a \cdot b) + (a \cdot c) \)
- **neutral elements**
  - \( \forall a. 0 + a = a \)
  - \( \forall a. 1 \cdot a = a \)
- **integrity law**
  - \( \forall a, b. a \cdot b = 0 \rightarrow a = 0 \lor b = 0 \)
- **partial order axioms**
  - \( \forall a. a \leq a \)
  - \( \forall a, b. a \leq b \land b \leq a \rightarrow a = b \)
  - \( \forall a, b, c. a \leq b \land b \leq c \rightarrow a \leq c \)
- **total order axiom**
  - \( \forall a, b. a \leq b \lor b \leq a \)
- **metric space**
  - \( \forall a, b. 0 \leq \delta(a, b) \)
  - \( \forall a, b. 0 = \delta(a, b) \leftrightarrow a = b \)
  - \( \forall a, b. \delta(a, b) = \delta(b, a) \)
  - \( \forall a, b, c. \delta(a, c) \leq \delta(a, b) + \delta(b, c) \)
- **monotonicity of addition and multiplication**
  - \( \forall a, b, c. a \leq b \rightarrow a + c \leq b + c \)
  - \( \forall a, b, c. a \leq b \rightarrow a \cdot c \leq b \cdot c \)
- **elimination rules**
  - \( \forall a, b, c. a + c \leq b + c \rightarrow a \leq b \)
  - \( \forall a, b, c. a + c = b + c \rightarrow a = b \)
  - \( \forall a, b, c. a \cdot \text{succ}(c) \leq b \cdot \text{succ}(c) \rightarrow a \leq b \)
  - \( \forall a, b, c. a \cdot \text{succ}(c) = b \cdot \text{succ}(c) \rightarrow a = b \)
- **modulo laws**
  - \( \forall a, b. (a \mod b) < b \)
  - \( \forall a, b. ((a \mod b) \mod b) = (a \mod b) \)
  - \( \forall a, b, c. ((a \mod (b \cdot c)) \mod b) = (a \mod b) \)
  - \( \forall a, b, c. ((a + b) \mod c) = (((a \mod c) + (b \mod c)) \mod c) \)
  - \( \forall a, b, c. ((a \cdot b) \mod c) = (((a \mod c) \cdot (b \mod c)) \mod c) \)
  - \( \forall a, b, c. ((a \cdot b + c) \mod b) = (c \mod b) \)
  - \( \forall a, b. (a \mod b) = a \text{ for } a < b \)
  - \( \forall a, b. (b \mod b) = 0 \)

Fig. B.2. Simple Properties of Natural Numbers
Moreover, we have \( q = (m \div n) \) and \( r = (m \mod n) \).

**Proof.** We first prove the existence with an induction proof. For \( m = 0 \), we find \( q := 0 \) and \( r := 0 \) to satisfy \( 0 = q \cdot n + r \) and \( r < n \) (note that \( n \neq 0 \)). Hence, the property holds for \( m = 0 \). Now, assume that the property holds for some natural number \( m \), i.e., we assume that for any number \( n \neq 0 \), there are numbers \( q, r \) so that \( m = q \cdot n + r \) and \( r < n \) hold. Hence, we have \( \text{succ}(m) = q \cdot n + \text{succ}(r) \) and \( \text{succ}(r) \leq n \). If \( \text{succ}(r) < n \) holds, the proof is already done, since we can use the numbers \( q \) and \( \text{succ}(r) \). Otherwise, we have \( \text{succ}(r) = n \), and we use the numbers \( \text{succ}(q) \) and \( 0 \) to prove our property.

Hence, we have shown the existence of the numbers \( q \) and \( r \) for all given \( m, n \) with \( n \neq 0 \). Now, assume we would have two solutions, i.e., we would have two different pairs \((q_1, r_1)\) and \((q_2, r_2)\) with

1. \( m = q_1 \cdot n + r_1 \), hence, \( r_1 = m - q_1 \cdot n \)
2. \( r_1 < n \)
3. \( m = q_2 \cdot n + r_2 \), hence, \( r_2 = m - q_2 \cdot n \)
4. \( r_2 < n \)

We prove that this is not possible: If \( r_1 = r_2 \) would hold, then we would have \( m - q_1 \cdot n = m - q_2 \cdot n \), which is equivalent to \( q_1 = q_2 \). Thus, our two pairs \((q_1, r_1)\) and \((q_2, r_2)\) would not be different, which was however our assumption. Therefore, we conclude that \( r_1 \neq r_2 \) must hold. Now assume that \( r_1 < r_2 \) holds. Then, \( r_2 - r_1 = (q_2 - q_1) \cdot n \), i.e., \( r_2 - r_1 \) is a multiple of \( n \), and as \( r_2 - r_1 > 0 \) holds, we moreover have \( r_2 - r_1 \geq n \) (since \( q_2 - q_1 \geq 1 \)). However, it then follows that \( r_2 \geq n + r_1 \), which contradicts \( r_2 < n \). Hence, we conclude that neither \( r_1 = r_2 \) nor \( r_1 < r_2 \) can hold. Analogously, we can prove that \( r_2 < r_1 \) can also not hold, and therefore it is impossible to have two different solutions \((q_1, r_1)\) and \((q_2, r_2)\).

Finally, we can prove by induction on \( m \) that the following properties are valid:

1. \( m = (m \div n) \cdot n + (m \mod n) \)
2. \( (m \mod n) < n \).

Having discussed the axioms, essential definitions, and properties of the natural numbers, we are now ready to develop more efficient algorithms to perform the arithmetic operations.

**B.2 Radix-\(B\) Numbers**

The definition of the natural numbers as given by the Peano axioms directly leads to the unary number system where the natural numbers are presented
as \( \text{succ}^n(0) \). While this is sufficient for theoretical investigations, it is not at all useful for the implementation of efficient arithmetic algorithms.

Radix number systems are a major improvement in comparison to the unary number system. Radix numbers are given in a more compact form, and the arithmetic operations can be directly performed on this compact representation which leads to improved runtimes for executing the arithmetic operations. In modern computer systems, almost all arithmetics for unsigned numbers is done in the radix-\( B \) number systems.

Radix numbers have already been used by the Babylonians that used radix numbers to the basis \( B = 60 \), but have neither been used by the Greeks nor by the Romans. In the following, we keep the considerations independent of the chosen radix \( B \), and consider the very special, but very important case \( B = 2 \) in Section B.5.

**Definition B.3 (Radix Number Systems).** Given a natural number \( B > 1 \) called the radix, every list \([x_{n-1}, \ldots, x_0]\) of natural numbers is mapped to a natural number \( \langle [x_{n-1}, \ldots, x_0] \rangle_B \) as follows:

\[
\langle [x_{n-1}, \ldots, x_0] \rangle_B := \sum_{i=0}^{n-1} x_i \cdot B^i
\]

The set of digits for radix \( B \) is \( \{0, \ldots, B-1\} \). A number \( \langle [x_{n-1}, \ldots, x_0] \rangle_B \) with digits \( x_i \in \{0, \ldots, B-1\} \) is called a radix-\( B \) number.

In the following, we often say that \( \langle [x_{n-1}, \ldots, x_0] \rangle_B \in \mathbb{N} \) is a radix-\( B \) number, which means that this natural number is given as a list of digits that are evaluated according to the above definition. This is justified by the following theorem that asserts that every natural number is uniquely represented by such a list of digits for any radix \( B \neq 0 \) except for leading zero digits:

**Theorem B.4 (Unique Representation).** The following facts hold for the radix-\( B \) number system:

- \( \langle [x_{n-1}, \ldots, x_0] \rangle_B = \langle [x_{n-1}, \ldots, x_1] \rangle_B \cdot B + x_0 \)
- Every natural number can be represented in the radix-\( B \) number system.
- A minimal number of digits \( \ell_x := \lfloor \log_B (x+1) \rfloor \) is required to represent a number \( x \).
- For every fixed \( \ell \geq \ell_x \), the number representation \( \langle [x_{\ell-1}, \ldots, x_0] \rangle_B \) of \( x \) is uniquely determined.
- If \( \ell_x \) is the minimal number of digits, then for all \( \ell \geq \ell_x \), we have

\[
x = \langle [x_{\ell-1}, \ldots, x_0] \rangle_B \text{ iff } \forall i \leq \ell. \ x_i = \left( (x \div B^i) \mod B \right).
\]

**Proof.** It can be easily seen that \( \langle [x_{n-1}, \ldots, x_0] \rangle_B = \left( \sum_{i=1}^{n-1} x_i \cdot B^{i-1} \right) \cdot B + x_0 \) holds which immediately proves the first proposition. It therefore follows by Theorem B.2 that
• \( (\lfloor x_{n-1}, \ldots, x_0 \rfloor_B \mod B) = x_0 \)
• \( (\lfloor x_{n-1}, \ldots, x_0 \rfloor_B \text{ div } B) = (\lfloor x_{n-1}, \ldots, x_1 \rfloor_B \)

Moreover, it follows by Theorem B.2 that \( x_0 \) is uniquely determined. Applying the same argument to \( (\lfloor x_{n-1}, \ldots, x_1 \rfloor_B \) and \( (\lfloor y_{n-1}, \ldots, y_1 \rfloor_B \) with the same number of digits are equal iff all of their digits are equal.

Given a number \( x \), we can therefore compute its radix-B number representation as follows: we determine \( x_0 := (x \mod B) \), and \( y_0 := (x \text{ div } B) \). By induction, we determine the radix-B number representation of \( y_0 \) in the same manner. Then, \( x \) has the radix-B number representation that is obtained by adding the digit \( x_0 \) to the right end of the radix-B number representation of \( y_0 \). Hence, it follows that every natural number can be represented as a radix-B number.

It is easily seen that adding zero digits at the left end of a radix-B number does not modify the represented number. Hence, every number has even infinitely many representations that differ by leading zeros. A canonical representation is the one without leading zeros.

For hardware implementations it is often necessary to consider radix-B numbers with a fixed number of digits. In this case, we can only represent a finite set of numbers that is described in the following lemma:

**Lemma B.5 (Fixed Number of Digits).** The set of natural numbers that is represented by \( n \)-digit radix-B numbers is of the form \( \{0, \ldots, \text{MaxNat}_B(n)\} \), where \( \text{MaxNat}_B(n) \) is the following natural number:

\[
\text{MaxNat}_B(n) = (\lfloor B-1, \ldots, B-1 \rfloor_B) = B^n - 1
\]

Moreover, we have the following special cases:

• \( (\text{MaxNat}_B(m + n) \text{ div } B^n) = \text{MaxNat}_B(m) \)
• \( (\text{MaxNat}_B(m + n) \text{ mod } B^n) = \text{MaxNat}_B(n) \).

**Proof.** The proof is simply based on the observation that an \( n \)-digit radix-B number is the larger the larger its digits are, and that \( \sum_{i=0}^{n-1} (B-1) \cdot B^i = (B-1) \cdot \sum_{i=0}^{n-1} B^i = B^n - 1 \) holds (geometric sum). The last two statements are seen as follows (note that \( \text{MaxNat}_B(n) < B^n \) holds and consider Theorem B.2):

\[
\sum_{i=0}^{m+n-1} (B-1) \cdot B^i = \left( \sum_{i=0}^{m-1} (B-1) \cdot B^i \right) \cdot B^n + \left( \sum_{i=0}^{n-1} (B-1) \cdot B^i \right)
\]

The summation rule of the geometric sum also leads to the well-known divisibility test in the radix-B number system by considering the digit sum:
Lemma B.6 (Digit Sum Divisibility Test).  
The radix-$B$ number $\langle [x_{n-1}, \ldots, x_0] \rangle_B^N$ is divisible by $B-1$ iff its digit $\sum_{i=0}^{n-1} x_i$ is divisible by $B-1$.

Proof. The geometric summation equation $(B-1) \cdot \sum_{i=0}^{n-1} B^i = B^n - 1$ tells us that the number $B^n - 1$ is divisible by $B-1$, since it is a product of $B-1$ and $\sum_{i=0}^{n-1} B^i$. Hence, it follows that for every $k$, the number $B^k - 1$ is divisible by $B-1$. Now, note that we can represent $x := \langle [x_{n-1}, \ldots, x_0] \rangle_B^N$ as the following sum:

$$
\sum_{i=0}^{n-1} x_i \cdot B^i = \sum_{i=0}^{n-1} x_i \cdot (B^i - 1 + 1) = \left( \sum_{i=0}^{n-1} x_i \cdot (B^i - 1) \right) + \left( \sum_{i=0}^{n-1} x_i \right).
$$

Since every number $B^k - 1$ is divisible by $B-1$, so is $y$. Hence, it follows that $(x \ mod \ (B-1)) = (z \ mod \ (B-1))$, and therefore $x$ is divisible by $B-1$ iff $z$ is divisible by $B-1$.

Of course, the calculation of the digit sum can be applied to the obtained digit sum as well until only a single digit is obtained. The result can also be generalized to the following higher radix representation:

Lemma B.7 (Higher Radix Representation).  
The radix-$B$ number $\langle [x_{n-1}, \ldots, x_0] \rangle_B^N$ can be converted to a radix-$B^k$ representation by simply grouping the digits of $x$ into bundles of $k$ digits, i.e., for $n = \ell \cdot k$, we obtain

$$
\langle [x_{\ell \cdot k-1}, \ldots, x_{(\ell-1) \cdot k}], [x_{\ell \cdot k+1-1}, \ldots, x_{\ell \cdot k}], \ldots, [x_{k-1}, \ldots, x_0] \rangle_B^{N_k}.
$$

Proof. The proof is very simple: We just group the summands in subsequent sums of $k$ summands starting at the lowest power $B^0$, i.e., for $n-1 = \ell \cdot (k-1)$, we obtain

$$
x = \sum_{i=0}^{n-1} x_i \cdot B^i = \sum_{j=0}^{\ell-1} \left( \sum_{i=0}^{k-1} x_{j \cdot k+i} \cdot B^i \right) \cdot (B^k)^j
= \langle [x_{j \cdot k+k-1}, \ldots, x_{j \cdot k}] \rangle_B^{N_k}
$$

Combining the previous two lemmata, we obtain also divisibility tests for division of radix-$B$ numbers by $B^k - 1$. For example, $x = \langle [2, 6, 7, 3, 9] \rangle_{10}^N$ is divisible by 99 since $26 + 73 + 99 = 198$ and moreover $1 + 98 = 99$ is divisible by 99.

Higher radix representations are often used in hardware implementations of arithmetic operations. As we will see below, the algorithms to implement the arithmetic operators contain loops that depend on the numbers of digits. Hence, the number of loop iterations decreases when a higher radix is chosen, but clearly, the digit arithmetics with higher radices becomes more complex.
In some cases, we consider expressions \( \langle x_{n-1}, \ldots, x_0 \rangle_B^N \) where the numbers \( x_i \) are outside the digit set \( \{0, \ldots, B-1\} \). The theorem below states that the algorithm shown in Figure B.3 can then be used to convert such a number to a radix-\( B \) number (with legal digits). If we apply this theorem to the extreme case \( \langle x_0 \rangle_B^N \) with an arbitrary natural number \( x_0 \), then we moreover see how we can convert any number \( x_0 \) to its radix-\( B \) representation.

**Theorem B.8 (Radix-B Conversion).** Given natural numbers \( x_{n-1}, \ldots, x_0 \), then the number \( \langle x_{n-1}, \ldots, x_0 \rangle_B^N = \sum_{i=0}^{n-1} x_i \cdot B^i \) can be converted to a radix-B number \( \langle y_{m-1}, \ldots, y_0 \rangle_B^N \) as shown in Figure B.3.

**Proof.** After iteration \( i \), the equation \( \langle x_{i}, \ldots, x_{0} \rangle_B^N = \langle c, y_{i}, \ldots, y_{0} \rangle_B^N \) is valid: for the induction base, we have \( \langle x_0 \rangle_B^N = x_0 = (x_0 \div B) \cdot B + (x_0 \mod B) = \langle c, y_0 \rangle_B^N \). For the induction step, we have the following:

\[
\begin{align*}
\langle x_{i+1}, \ldots, x_0 \rangle_B^N &= x_{i+1} \cdot B^{i+1} + \langle x_{i}, \ldots, x_0 \rangle_B^N \\
&= x_{i+1} \cdot B^{i+1} + \langle c, y_{i}, \ldots, y_{0} \rangle_B^N \\
&= x_{i+1} \cdot B^{i+1} + c \cdot B^{i+1} + \langle y_{i}, \ldots, y_{0} \rangle_B^N \\
&= (x_{i+1} + c) \cdot B^{i+1} + \langle y_{i}, \ldots, y_{0} \rangle_B^N \\
&= \left((x_{i+1} + c) \div B \right) \cdot B + \left((x_{i+1} + c) \mod B \right) + \langle y_{i}, \ldots, y_{0} \rangle_B^N \\
&= \langle y_{i+1} \rangle_B^N \\
\end{align*}
\]

By definition, the new digit \( y_{i} \) in the algorithm is always a legal digit, since it is determined as the remainder of a division by \( B \). However, the carry \( c \) may be arbitrarily large, and therefore the final carry has to be converted by the
same procedure after the for-loop with another loop. Note that the number of
digits that have to be determined by this loop depends on the size of $c$, and
that the loop will finally terminate since $c$ will decrease in each iteration.

Based on Theorem B.8, we can already perform some simple arithmetic
operations. For example, it is easily seen that the following equations hold:

\begin{itemize}
\item $\langle [x_{n-1}, \ldots, x_0]_B \rangle_B + \langle [y_{n-1}, \ldots, y_0]_B \rangle_B = \langle [x_{n-1} + y_{n-1}, \ldots, x_0 + y_0]_B \rangle_B$
\item $\langle [x_{n-1}, \ldots, x_0]_B \rangle_B - \langle [y_{n-1}, \ldots, y_0]_B \rangle_B = \langle [x_{n-1} - y_{n-1}, \ldots, x_0 - y_0]_B \rangle_B$
\item $\langle [x_{n-1}, \ldots, x_0]_B \rangle_B \cdot y = \langle [x_{n-1} \cdot y_0, \ldots, x_0 \cdot y]_B \rangle_B$
\end{itemize}

However, while the given operands may consist of legal digits, the resulting
numbers may not be radix-$B$ numbers, since their entries may not belong to
the digit set $\{0, \ldots, B - 1\}$. However, by Theorem B.8 we can easily convert
these numbers to legal radix-$B$ numbers. This yields directly the addition and
subtraction algorithms shown in Figure B.4.

The above equations show that we can postpone the conversion to the
legal digit set, and may work with larger integers instead. This will be useful
for some of the algorithms in the following, but it has to be used with some
care in practice to make sure that the intermediate values do not lead to
overflows when the ‘digit’ arithmetics is performed as machine instructions of
a processor.

**Theorem B.9 (Addition and Subtraction of Radix-$B$ Numbers).**

Given radix-$B$ numbers $x = \langle [x_{n-1}, \ldots, x_0]_B \rangle_B$ and $y = \langle [y_{n-1}, \ldots, y_0]_B \rangle_B$, and
the results $(c_a, [a_{n-1}, \ldots, a_0]) = \text{NatAdd}(x, y)$ and $(c_b, [b_{n-1}, \ldots, b_0]) = \text{NatSub}(x, y)$ of the functions \text{NatAdd} and \text{NatSub} shown in Figure B.4, then the
following holds:

\begin{itemize}
\item $c_a \in \{0, 1\}$
\item $x + y = \langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B$
\item $\langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B$ is a radix-$B$ number
\item $c_b \in \{-1, 0\}$
\item if $c_b = 0$, then $x - y = \langle [c_b, b_{n-1}, \ldots, b_0] \rangle_B$
\item if $c_b = -1$, then $x < y$
\item $\langle [b_{n-1}, \ldots, b_0] \rangle_B$ is a radix-$B$ number
\end{itemize}

**Proof.** For the proof, note that $\langle [x_{n-1}, \ldots, x_0]_B \rangle_B + \langle [y_{n-1}, \ldots, y_0]_B \rangle_B = \langle [x_{n-1} + y_{n-1}, \ldots, x_0 + y_0]_B \rangle_B$ holds. Hence, $x + y = \langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B$ follows from
Theorem B.8. Equation $x - y = \langle [c_b, b_{n-1}, \ldots, b_0] \rangle_B$ is proved analogously.

To prove that the results contain legal digits, we first prove that in the
addition algorithm the variable $c$ has either the value 0 or 1: This is initially
the case, and after each iteration, it holds, since $x_i + y_i + c_i \leq 2 \cdot (B - 1) + 1 = B + (B - 1) = (1, B - 1)_B$ holds, which means that the maximal sum digit is $B - 1$ and the maximal carry is 1. Hence, we can represent the sum $x_i + y_i + c_i$ with the two digits $c_{i+1}, s_i$. 
function NatAdd(int x[n], y[n]) {
    long int sm;
    int s[n],c;
    c=0;
    for(int i=0..n-1) {
        sm = x[i] + y[i] + c;
        s[i] = sm mod B;
        c = sm div B;
    }
    return (c,s);
}

function NatSub(int x[n], y[n]) {
    long int sm;
    int s[n],c;
    c=0;
    for(int i=0..n-1) {
        sm = x[i] - y[i] + c;
        s[i] = sm mod B;
        c = sm div B;
    }
    return (c,s);
}

Fig. B.4. Addition and Subtraction of Radix-\(B\) Numbers

Similarly, the variable \(c\) in the subtraction algorithm is either -1 or 0 which is again seen by induction: We have \(c = 0\) before the iteration and after each iteration, we estimate on the one hand \(x_i - y_i + c_i \leq (B - 1) = \langle 0, B - 1 \rangle_B\) and on the other hand \(x_i - y_i + c_i \geq -(B - 1) - 1 = -B = \langle [-1, 0] \rangle_B\).

Finally, note that the final carry is -1 iff \(x < y\) holds (otherwise the sum would be positive).

The subtraction function allows us also to check whether \(x < y\) holds, since this information is reported on the final carry. Of course, the comparison can also be directly implemented, which is done as follows:

**Theorem B.10 (Comparison of Radix-\(B\) Numbers).**

Given numbers \(x = \langle [x_{n-1}, \ldots, x_0] \rangle_B^N\) and \(y = \langle [y_{n-1}, \ldots, y_0] \rangle_B^N\), the following holds:

- \(x = y\) iff \(x_i = y_i\) for all \(i \in \{0, \ldots, n-1\}\)
- \(x < y\) iff \(x_i < y_i\) holds for the largest index \(j\) where \(x_j \neq y_j\)

**Proof.** The proof of the first proposition directly follows from the uniqueness of the representation with the same number of digits. The second proposition is also easy to prove: Assume \(x\) and \(y\) have the following representations:
B.2 Radix-\(B\) Numbers

function NatLess(int x[n], y[n]) {
    int i = n - 1;
    while (x[i] == y[i] & i >= 0) i = i - 1;
    if (i < 0) return false;
    else return x[i] < y[i];
}

function NatEqual(int x[n], y[n]) {
    int i = n - 1;
    while (x[i] == y[i] & i >= 0) i = i - 1;
    return (i < 0);
}

Fig. B.5. Comparison of Radix-\(B\) Numbers

- \(x = \langle [z_{n-1}, \ldots, z_{i+1}, x_i, \ldots, x_0] \rangle_B^N\) and
- \(y = \langle [z_{n-1}, \ldots, z_{i+1}, y_i, \ldots, y_0] \rangle_B^N\),
- where \(x_i \neq y_i\) holds.

This means that the radix-\(B\) representations of \(x\) and \(y\) start with a common prefix \([z_{n-1}, \ldots, z_{i+1}]\). Then, we obviously have

\[
y - x = \langle [y_i, \ldots, y_0] \rangle_B^N - \langle [x_i, \ldots, x_0] \rangle_B^N
= (y_i \cdot B^i + \langle [y_{i-1}, \ldots, y_0] \rangle_B^N) - (x_i \cdot B^i + \langle [x_{i-1}, \ldots, x_0] \rangle_B^N)
= (y_i - x_i) \cdot B^i + \langle [y_{i-1}, \ldots, y_0] \rangle_B^N - \langle [x_{i-1}, \ldots, x_0] \rangle_B^N
\]

Now, we prove that \(y - x > 0\) iff \(y_i - x_i > 0\). By Lemma B.5, we know that \(-(B^i - 1) \leq w \leq B^i - 1\). Therefore, \(y - x\), i.e., \(u + w\) is positive iff \(y_i - x_i\) is positive, since \(u\) is so positive or so negative that the addition of a value in the range of \(w\) can not change the sign of \(u\).

\[\square\]

Hence, we can use the functions given in Figure B.5 to implement a test on comparison and equality of radix-\(B\) numbers, respectively. It should be clear how the other comparison operations like \(>\), \(\le\), \(\ge\) can be implemented.

It is remarkable that the functions in Figure B.5 suggest that the comparison should be tested by checking the digits from the most to the least significant digits. However, considering the subtraction function of Figure B.4, it is clear that this can also be done by checking the digits from the least to the most significant digit. This is important if the arithmetic is done serially like in case of the automata representation of Presburger arithmetic.

For fixed numbers of digits, the subtraction operation can alternatively be implemented by adding a complement of the second operand, which is defined as follows:
Moreover, for \(x\) represented with digits, so that \(x\) may be viewed as the negative value of \(x\). Indeed, we will see that adding \(x\) will have essentially the same effect as subtracting \(x\).

Since our definition of \(x\) does not directly lead to an algorithm to compute \(x\) on the radix-\(B\) representation of \(x\), we first prove the following theorem that allows us to do this:

**Theorem B.12 (Complement of Radix-\(B\) Numbers).** For any radix-\(B\) number \(x = \langle [x_{n-1}, \ldots, x_0] \rangle_B\), we have \(x^c := B^n - x\) as its complementary number. By definition, the sum \(x + x^c\) is always \(B^n\) for all \(n\)-digit numbers. If we deal with \(n\)-digit numbers, and ignore overflows, then \(B^n\) corresponds with zero, so that \(x^c\) may be viewed as the negative value of \(x\). Indeed, we will see that adding \(x^c\) will have essentially the same effect as subtracting \(x\).

Proof. Let us abbreviate \(x' := \langle [B - 1 - x_{n-1}, \ldots, B - 1 - x_0] \rangle_B + 1\), so that we have to prove that \(x^c = x'\) holds. We prove this by showing that \(x + x' = B^n\) holds:

\[
\langle [x_{n-1}, \ldots, x_0] \rangle_B + \langle [B - 1 - x_{n-1}, \ldots, B - 1 - x_0] \rangle_B + 1 \\
= \left( \sum_{i=0}^{n-1} x_i \cdot B^i \right) + \left( \sum_{i=0}^{n-1} (B - 1 - x_i) \cdot B^i \right) + 1 \\
= \left( \sum_{i=0}^{n-1} (x_i + B - 1 - x_i) \cdot B^i \right) + 1 \\
= \text{MaxNat}_B(n) + 1 \\
= (B^n - 1) + 1 = B^n
\]

Finally, note that the addition \(\langle [B - 1 - x_{n-1}, \ldots, B - 1 - x_0] \rangle_B + 1\) requires a further digit if and only if the initial carry propagates to the most significant digit. This is only the case for \(x = 0\).

Hence, the complement \(x^c\) can be computed on the radix-\(B\) representation of \(x\). This leads to the following subtraction algorithm that works by adding the complement of the second summand:

**Theorem B.13 (Subtraction by Addition of the Complement).**

Given numbers \(x = \langle [x_{n-1}, \ldots, x_0] \rangle_B\), \(y = \langle [y_{n-1}, \ldots, y_0] \rangle_B\), we define the following sequence of digits with \(c_0 := 1\):

- \(s_i := ((x_i + (B - 1 - y_i) + c_i) \mod B)\) and
- \(c_{i+1} := ((x_i + (B - 1 - y_i) + c_i) \div B)\)

for \(i \in \{0, \ldots, n - 1\}\). Then, we have the following:

- \(c_i \in \{0, 1\}\) for all \(i\)
function Compl(int x[n]) {
    int sm,y[n],c;
    c=1;
    for(int i=0..n-1) {
        sm = (B - 1 - x[i]) + c;
        y[i] = sm mod B;
        c = sm div B;
    }
    return (c,y);
}

function NatSub(int x[n], y[n]) {
    long int sm;
    int s[n],c;
    c=1;
    for(int i=0..n-1) {
        sm = x[i] + (B - 1 - y[i]) + c;
        s[i] = sm mod B;
        c = sm div B;
    }
    return (c,s);
}

Fig. B.6. Complement and Subtraction of Radix-$B$ Numbers

- $B^n + x - y = \langle[c_n, s_{n-1}, \ldots, s_0]\rangle_B^N$
- $y \leq x$ if and only if $c_n > 0$ (which is equivalent to $c_n = 1$)
- if $c_n > 0$ (i.e., $c_n = 1$), then $x - y = \langle[s_{n-1}, \ldots, s_0]\rangle_B^N$

Proof. It is easily seen that instead of computing the digits $c_n$ and $[s_{n-1}, \ldots, s_0]$ as stated in the theorem, we can alternatively add the complement of $y$. The resulting digits $c_n$ and $[s_{n-1}, \ldots, s_0]$ are then computed exactly as required in the above theorem. Because of Theorem B.9, it therefore follows that $x + y^c = \langle[c_n, s_{n-1}, \ldots, s_0]\rangle_B^N$ holds. By definition of the complement, it follows that $x + y^c = x + B^n - y$, and therefore, we have (1) $x - y + B^n = \langle[c_n, s_{n-1}, \ldots, s_0]\rangle_B^N$.

The extreme cases $\max\text{Nat}_B(n) - 0$ and $0 - \max\text{Nat}_B(n)$ show that $-(B^n - 1) \leq x - y \leq B^n - 1$ holds. Adding $B^n$ allows us to derive the estimations (2) $1 \leq x - y + B^n \leq 2 \cdot B^n - 1$, which show that the result $x - y$ can be represented with $n$ digits for $y \leq x$.

It remains to check whether the result $x - y$ is not negative and how we find the $n$ digits to represent $x - y$ in this case. To this end, note that $y \leq x$ is equivalent to $B^n \leq x - y + B^n$. By (1), it now follows that $y \leq x$ is equivalent to $B^n \leq \langle[c_n, s_{n-1}, \ldots, s_0]\rangle_B^N$, and thus, (4) $y \leq x$ is equivalent to $c_n > 0$.

Finally, if $0 < c_n$ holds, we know by (2) and (4) that $0 \leq x - y \leq B^n - 1$ holds. By (1) and (2), we can moreover conclude that (5) $c_n \in \{0, 1\}$ holds,
since \( c_n = ((x - y + B^n) \div B^n) \) due to (1). Hence, if \( 0 < c_n \) holds, we have \( c_n = 1 \), and therefore, \( x - y = \langle [c_n, s_{n-1}, \ldots, s_0] \rangle_B^N - B^n \), i.e., \( x - y = \langle [s_{n-1}, \ldots, s_0] \rangle_B^N \).

Hence, for fixed numbers of digits, we can alternatively implement the subtraction operation by adding the complement. However, this does not really perform the subtraction operation, and therefore it is not convenient when working with arbitrarily precise numbers.

```c
function NatMult(int x[m], y[n]) {
    long int s;
    int p[m+n], c;
    for(int i=0..n-1) {
        // loop body assigns p[m+i..i] = p[m+i-1..i] + x[m-1..0] * y[i];
        c=0;
        for(int j=0..m-1) {
            s = p[i+j] + x[j] * y[i] + c;
            p[i+j] = s mod B;
            c = s div B;
        }
        p[i+m] = c;
    }
    return p;
}
```

Fig. B.7. Multiplication of Radix-\( B \) Numbers

Multiplication of radix-\( B \) numbers can be done as with the usual paper-and-pencil method. To this end, let us first observe that the product of an \( m \)-digit number with an \( n \)-digit number may have up to \( m + n \) digits: The worst case is, of course, \( \text{MaxNat}_B(m) \cdot \text{MaxNat}_B(n) \), i.e., \( (B^m-1) \cdot (B^n-1) = B^{m+n} - (B^m + B^n - 1) < B^{m+n} - 1 \). Although \( \text{MaxNat}_B(m+n) \) is not the product of radix-\( B \) numbers with \( m \) and \( n \) digits, we still require \( m + n \) digits, since these are necessary for the above product.

**Theorem B.14 (Multiplication of Radix-\( B \) Numbers).**

*Given numbers \( x = \langle [x_{m-1}, \ldots, x_0] \rangle_B^N \) and \( y = \langle [y_{n-1}, \ldots, y_0] \rangle_B^N \), the algorithm shown in Figure B.7 computes the \( m + n \)-digit number \( p \) so that \( x \cdot y = p \) holds.*

**Proof.** The crucial idea of the algorithm is to simply sum up so-called partial products as follows:

\[
x \cdot y = x \cdot \sum_{i=0}^{n-1} y_i \cdot B^i = \sum_{i=0}^{n-1} x \cdot y_i \cdot B^i.
\]
The multiplication with \( B^i \) can be done by shifting the products \( y_i \cdot x \) by \( i \) digits as done in Figure B.7, where the inner loop computes the assignment \( p[m+i..i] = p[m+i-1..i] + x[m-1..0] \cdot y[i] \).

It remains to check that the computed digits are in the required interval: Note that the intermediate value \( s \) that is computed in the inner loop may have at most size \( 2(B - 1) + (B - 1)^2 = B^2 - 1 = (B - 1, B - 1) \), and therefore, both \( p[i+j] \) and \( c \) are in the range of digits and \( s \) is in the range of a double digit.

The multiplication algorithm given in Figure B.7 contains two nested loops, so that its runtime is \( O(mn) \) for given \( m \)- and \( n \)-digit numbers \( x \) and \( y \). The algorithm is essentially the well-known paper-and-pencil method that we all have learned in school.

Note that for an implementation of the algorithm the arithmetic on digit numbers must be capable to compute values of size \( B^2 - 1 \) (for the assignment to variable \( s \)).

Note further that after the \( i \)-th iteration of the outer loop, we already have computed the least \( i \) digits of the product, since the digits \( p[i-1..0] \) will not be overwritten after this iteration. Hence, in an application where the digits have to be sent out serially, we could start doing this in parallel with the computation.

It should be remarked here that there are more efficient algorithms for multiplication. For example, the simple alternative of the Karatsuba algorithm [140] yields a much better asymptotic complexity. This algorithm is best described for multiplication of numbers \( x \) and \( y \) with \( 2n \) digits. Assume that \( 2^n = 2k \), then the radix-\( B \) number representations of \( x \) and \( y \) are halved, so that \( x = x_1 \cdot B^k + x_0 \) and \( y = y_1 \cdot B^k + y_0 \) holds. Clearly, we could compute the product \( x \cdot y \) as \( (x_1 \cdot B^k + x_0) \cdot (y_1 \cdot B^k + y_0) = x_1 \cdot y_1 \cdot B^{2k} + (x_0 \cdot y_1 + x_1 \cdot y_0) \cdot B^k + x_0 \cdot y_0 \), and we could repeat this divide-and-conquer method also for computing the four subsequent products \( x_1 \cdot y_1, x_0 \cdot y_1, x_1 \cdot y_0, \) and \( x_0 \cdot y_0 \). The resulting algorithm would have the same runtime as our algorithm in Figure B.7. However, we can compute the product also with only three subsequent products \( p_0 := x_0 \cdot y_0, p_1 := x_1 \cdot y_1, p_2 := (x_1 + x_0) \cdot (y_1 + y_0) \) as follows:

\[
x \cdot y = p_1 \cdot B^{2k} + (p_2 - (p_0 + p_1)) \cdot B^{2k} + p_0
\]

A simple complexity analysis then reveals that the runtime to compute the product of two \( n \)-digit numbers reduces from \( O(n^2) \) to \( O(n \log_2(3)) \). The Karatsuba algorithm is based on the observation that multiplication is more expensive than addition, and therefore a recursive call trades addition operations against multiplication operations.

However, also the Karatsuba method is far from being optimal. The best known algorithm for multiplication of \( n \)-digit numbers is the Schönhage-Strassen algorithm [235] that only requires time \( O(n \cdot \log(n) \cdot \log(\log(n))) \).
Finally, let us consider the division of radix-$B$ numbers. Note that algorithms for division also compute the modulo operation due to the tight relationship $(x \mod y) = x - (x \div y) \cdot y$ between division and modulo operations.

Again, we follow the usual paper-and-pencil method to derive a first version of the radix-$B$ division of a $m$-digit number $x$ by a $n$-digit number $y$. Let us first estimate bounds for the quotient and the remainder, so that we know how many digits we have to provide in the worst case. Since division is monotonic, the largest quotient is obtained by dividing the largest $x$ by the smallest $y$, i.e., by dividing $\text{MaxNat}_B(m)$ by 1, which yields the quotient $\text{MaxNat}_B(m)$ with $m$-digits. By definition, the remainder is a number that is smaller than $y$, and therefore $n$ digits will be sufficient for the remainder. Hence, if we allow leading zero digits, then for dividing an $m$ digit number $x$ by a $n$ digit number $y$, we need $m$ digits for the quotient and $n$ digits for the remainder.

Similar to the multiplication algorithm, we deal with partial products. However, instead of adding them, we subtract successively the numbers $q_i \cdot y \cdot B^i$ that stem from the radix-$B$ representation of the quotient. This is easily seen as follows:

$$r = x - y \cdot q = x - y \cdot \sum_{i=0}^{m-1} q_i \cdot B^i = x - q_{m-1} \cdot y \cdot B^{m-1} - q_{m-2} \cdot y \cdot B^{m-2} - \ldots\tag{4}$$

These considerations do not already lead to an algorithm for the division operation, since we have to determine the quotient digits. However, the next lemma shows that the quotient digits are determined by the above equation when we demand that they are maximally chosen in the following sense:

**Lemma B.15 (Division of Radix-$B$ Numbers).** Given the radix-$B$ numbers $x = \langle [x_{m-1}, \ldots, x_0] \rangle_B^N$ and $y = \langle [y_{n-1}, \ldots, y_0] \rangle_B^N$ with $y \neq 0$, we define the $m+1$ radix-$B$ numbers $x^{(i)}$ recursively as follows:

- $x^{(m)} := x$
- $x^{(i-1)} := x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$ for $i = m, \ldots, 1$, where $q_{i-1}$ is the maximal digit $\in \{0, \ldots, B-1\}$ with $q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)},$ for all $i$, we have the invariants $(I_1)$ $x^{(i)} + \langle [q_{m-1}, \ldots, q_i] \rangle_B^N \cdot y \cdot B^i = x$ and $(I_2) x^{(i)} < y \cdot B^i$. Therefore, the quotient $q$ and the remainder $r$ can be obtained as the radix-$B$ numbers $q = \langle [q_{m-1}, \ldots, q_0] \rangle_B^N$ and $r = x^{(0)}$.

**Proof.** We first prove invariant $I_1$: For $i = m$, we obviously have $x^{(m)} + \langle [0] \rangle_B^N \cdot y \cdot B^m = x^{(m)} = x$. For smaller $i$, we obtain $x^{(i-1)} = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1} \equiv \langle [q_{m-1}, \ldots, q_i] \rangle_B^N \cdot y \cdot B^i - q_{i-1} \cdot y \cdot B^{i-1} = x - \langle [q_{m-1}, \ldots, q_{i-1}] \rangle_B^N \cdot y \cdot B^{i-1},$ so that invariant $(I_1)$ holds.

Invariant $(I_2)$ is seen as follows: For $i = m$, we have $x^{(m)} = x \leq \text{MaxNat}_B(m) = B^m - 1 < B^m \leq y \cdot B^m$, so that $(I_2)$ already holds in case $i = m$. By definition, we have $x^{(i-1)} = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$, i.e.,
B.2 Radix-$B$ Numbers

(1) $x^{(i)} = x^{(i-1)} + q_{i-1} \cdot y \cdot B^{i-1}$. Since $q_{i-1}$ has been maximally chosen for the computation of $x^{(i-1)}$, we conclude (2) $q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)}$ and especially (3) $x^{(i)} < (q_{i-1} + 1) \cdot y \cdot B^{i-1}$. By (1) and (3), it follows that $x^{(i-1)} + q_{i-1} \cdot y \cdot B^{i-1} < (q_{i-1} + 1) \cdot y \cdot B^{i-1}$ holds, and therefore, we have $x^{(i-1)} < y \cdot B^{i-1}$. Hence, also $(I_2)$ holds.

Due to $(I_1)$ and $(I_2)$, we have in particular $x^{(0)} + \langle [q_{m-1}, \ldots, q_0] \rangle_B \cdot y = x$ and $x^{(0)} < y \cdot B^i$, so that by Theorem B.2, we conclude that $(x \div y) = \langle [q_{m-1}, \ldots, q_0] \rangle_B$ and $(x \mod y) = x^{(0)}$.

\[\Box\]

function DgtSub(digit z[n+1], q, y[n]) {
  // compute s[n..0] = z[n..0] - q * y[n-1..0]
  digit cs, sm, s[n+1];
  cs = 0;
  for(int i=0..n-1) {
    sm = cs + z[i] - q * y[i];
    s[i] = sm mod B;
    cs = sm div B;
  }
  s[n] = (cs + z[n]);
  return s;
}

function NatDivMod0(digit x[m], y[n]) {
  digit z[m+n], s[n+1], q[m], r[n];
  z[m-1..0] = x[m-1..0];
  for(int i=m-1..0) {
    // try digits B-1,B-2,...,0 for q[i]
    q[i] = B;
    do {
      q[i] = q[i] - 1;
      s[n..0] = DgtSub(z[n+i..i], q[i], y[n-1..0]);
      // now s[n..0] = z[n+i..i] - q[i] * y[n-1..0]
    } while(s[n]==0); // repeat if z[n+i..i] < q[i] * y[n-1..0]
    z[n+i..i] = s[n..0];
  }
  r[n-1..0] = z[n-1..0];
  return (q, r);
}

Fig. B.8. First Version of Division of Radix-$B$ Numbers

The previous lemma is the core of any 'subtractive' division algorithm. It almost leads directly to the preliminary algorithm shown in Figure B.8, which is however quite wasteful and inefficient. In this algorithm, the values $x^i$
mentioned in the previous lemma are stored in the variable \( z \) of function \( \text{NatDivMod0} \) after each iteration of the for-loop: initially, this is the case by initialization of \( z \), and after each iteration, this is the case since \( z \) is defined as the number \( s \) that is the result of a function call to function \( \text{DgtSub} \) described below.

It is easily seen that the function \( \text{DgtSub} \) is thereby used to compute for a given \( n+1 \) digit number \([z_n, \ldots, z_0]\), a digit \( q \), and a \( n \) digit number \([y_{n-1}, \ldots, y_0]\), a \( n+1 \) digit number \([s_n, \ldots, s_0]\) such that the following holds:

\[
\langle [s_n, \ldots, s_0] \rangle_B = \langle [z_n, \ldots, z_0] \rangle_B - q \ast \langle [y_{n-1}, \ldots, y_0] \rangle_B.
\]

Note that the result of the function call to \( \text{DgtSub} \) is not always a radix-\( B \) number, since the most significant digit may be negative: \(-(B - 1) \leq x_n \leq B - 1\).
Moreover, we have \( s_n \geq 0 \) iff \( \langle [z_n, \ldots, z_0] \rangle_B \geq q \ast \langle [y_{n-1}, \ldots, y_0] \rangle_B \).
Therefore, if the do-while loop in function \( \text{NatDivMod0} \) terminates, we have found a radix-\( B \) number \( s \) and the maximal digit \( q \), so that (1) \( \langle [s_n, \ldots, s_0] \rangle_B = \langle [z_n + i, \ldots, z_i] \rangle_B - q_i \ast \langle [y_{n-1}, \ldots, y_0] \rangle_B \), (2) \( s_n = 0 \), and (3) \( \langle [s_n, \ldots, s_0] \rangle_B \geq 0 \) holds.
Note further that we only need to consider the digits \([z_{n+i}, \ldots, z_i]\) of \( z \), since the digits \([z_n, \ldots, z_{n+i+1}]\) are all zero due to invariant \( I_2: x^{(i)} < y \cdot B^i < B^{n+i} \).
Furthermore, there is no need to consider the digits \([z_{i-1}, \ldots, z_0]\), since these are not affected by the subtraction of \( y \cdot B^i \).
Finally, note that we try digits \( q_i = B - 1, B - 2, \ldots \) for subtraction \( \langle [z_{n+i}, \ldots, z_i] \rangle_B - q_i \ast \langle [y_{n-1}, \ldots, y_0] \rangle_B \) until the result is a non-negative number (so that the desired digit \( q_i \) has been found).
Note that the result of \( \text{DgtSub} \) with an estimation of the digit is assigned to a local variable \( s \) so that we do not overwrite \( z \) when a wrong value has been chosen.
For this reason, we have to explicitly transfer the digits from \( s \) to \( z \) after we have found the correct quotient bit \( q[i] \) after the do-loop.
Finally, note that the initial variables \( x \) and \( y \) are not overwritten.

Hence, we have already found an algorithm for the division of radix-\( B \) numbers. However, this algorithm is not very efficient, so that we consider some optimizations in the following. The first improvement is that we eliminate the local variable \( z \) used in in Figure B.8. Looking closely at the algorithm of Figure B.8, one easily reveals that we only need the \( n+1 \) digits \([z_{n+i}, \ldots, z_i]\) in iteration \( i \) for the function call to \( \text{DgtSum} \).
Moreover, at this stage, we have \( z_i = x_i \), so that the last digit \( z_i \) could be borrowed directly from \( x \).
Hence, only the \( n \) digits \([z_{n+i}, \ldots, z_{i+1}]\) have to be stored somewhere (the higher digits of \( z \) are all zero, and the lower ones are identical to the corresponding ones of \( x \)).
Now, note that the \( n \) digits of the remainder \( r \) are only used at the end of the algorithm. Hence, it is natural to use the \( n \) digits of \( r \) during the computation to store the \( n \) digits \([z_{n+i}, \ldots, z_{i+1}]\).
This directly leads to the improved algorithm shown in Figure B.9, which is identical to Figure B.8 except for the elimination of \( z \) and the expansion of the function call to \( \text{DgtSum} \).
Figure B.10 shows a version of the same algorithm where the call to \( \text{DgtSub} \) has been expanded.

Note that this algorithm stores the values \( x^{(i)} \) in \( rt \{n-1..0]@x[i-1..0] \), i.e., the upper \( n \) digits in \( x \), and the lower \( i \) digits are
function NatDivMod1(digit x[m], y[n]) {
    digit s[n+1], q[m], r[n];
    for(int i=m-1..0) {
        // x(i+1) is stored in r[n-1..0]@x[i..0]
        q[i] = B;
        do {
            q[i] = q[i] - 1;
            // compute x(i) = s[n..0]@x[i-1..0] = x(i+1) - q[i] * y * B**i, i.e.
            // compute s[n..0] = r[n-1..0]@x[i] - q[i] * y[n-1..0]
            s[n..0] = DgtSub(r[n-1..0]@x[i], q[i], y[n-1..0]);
        } while(s[n]!=0);
        // finally, move s to r, note s[n]==0
        r[n-1..0] = s[n-1..0];
    }
    return (q,r);
}

Fig. B.9. Improved Version of Division of Radix-$B$ Numbers

taken from the original $x$. Initially ($i = m$), this holds since the digits $r_i$ are all zero, so that $x^{(m)} = x$ holds. In the iterations $i = m-1, \ldots, 0$, we then compute $x^{(i-1)} := x^{(i)} - q_d \cdot y \cdot B^{i-1}$ as follows:

\[
\begin{array}{cccccccc}
  r_{n-1} & r_{n-2} & \ldots & r_0 & x_{i-1} & x_{i-2} & \ldots & x_0 \\
  q_i & y_{n-1} & \ldots & y_1 & y_0 \\
  s_n & s_{n-1} & \ldots & s_1 & s_0 & x_{i-2} & \ldots & x_0 \\
  r_{n-1} & r_1 & r_0 & x_{i-2} & \ldots & x_0 \\
\end{array}
\]

Note that $y$ is shifted by $i - 1$ digits to the left to simulate the multiplication of $y$ with $B^{i-1}$. If the result $\langle s_n, \ldots, s_0 \rangle_B$ is negative (i.e., $s_n \neq 0$), then we repeat the above subtraction with a decreased quotient digit $q$. Otherwise, we have found the maximal $q_i$ and proceed with $\langle r_{n-1}, \ldots, r_0 \rangle := [s_{n-1}, \ldots, s_0]$ (omitting the zero digit $s_n$).

The previous algorithms tried the digits $B - 1, B - 2, \ldots, 0$ one after the other until a non-negative result has been obtained from the call to function DgtSub. At this stage, we have obtained the correct quotient digit and partial remainder for the further division. However, these algorithms have the disadvantage that we have to subtract a scaled divisor (as done by DgtSub). For efficient hardware implementations, it would be desirable to only work with addition and subtraction. This can be done when we try the digits $0, 1, \ldots, B - 2, B - 1$ in increasing order instead of decreasing order. As a result, we first obtain the restoring division algorithm shown in Figure B.11.

As the previous algorithm, the restoring division algorithm shown in Figure B.11 stores the partial remainders $x^{(i)}$ in $x[n-1..0]@x[i-1]$ and subtracts $y[n-1..0]$ from this number until the result becomes negative or the maximal
function NatDivMod1(digit x[m], y[n]) {
    digit s[n+1], q[m], r[n], sm, cs;
    for(int i=m-1..0) {
        // try digits B-1,B-2,...,0 for q[i]
        // x(i+1) is stored in r[n-1..0]@x[i..0]
        q[i] = B;
        do {
            q[i] = q[i] - 1;
            // compute x(i) = s[n..0]@x[i] = x(i+1) - q[i] * y * B^i, i.e.
            // compute s[n..0] = r[n-1..0]@x[i] - q[i] * y[n-1..0]
            sm = x[i] - q[i] * y[0];
            s[0] = sm mod B;
            cs = sm div B;
            for(int j=0..n-2) {
                sm = cs + r[j] - q[i] * y[j+1];
                s[j+1] = sm mod B;
                cs = sm div B;
            }
            sm = cs + r[n-1];
            s[n] = sm mod B;
        } while(s[n]!=0);
        // finally, move s to r, note s[n]==0
        for(int j=0..n-1)
            r[j] = s[j];
    }
    return (q,r);
}

Fig. B.10. Expanded Version of the Division Algorithm of Figure B.9

digit q[i]==B-1 is reached. This is done by first assigning r[n-1..0]@x[i-1] to s[n..0] and then by iteratively subtracting y[n-1..0] from s[n..0] until s becomes negative or q[i]==B-1 holds. If s should be negative, the last subtraction of y[n-1..0] was too much, and we have to restore the previous value of s. However, as this value is nowhere available, we have to recompute it by adding y[n-1..0] to s[n..0]. Then, we can finally assign the correct partial remainder s[n..0] to r[n-1..0] (note that then s[n]==0 holds).

Clearly, the restoring division algorithm has the disadvantage that in most cases, namely in those where the correct quotient digit is not B-1, it performs an additional addition step to restore the partial remainder. This can be circumvented by the nonrestoring division algorithm shown in Figure B.12.

The idea of the nonrestoring division algorithm is as follows: Recall that we should compute the partial remainder x^{(i-1)} := x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}. However, trying the quotient digits in increasing order might end with the value u^{(i-1)} := x^{(i)} - (q_{i-1} + 1) \cdot y \cdot B^{i-1} where the quotient digit is increased
function RestoreDivMod(digit x[m], y[n]) {
    digit s[n+1], q[m], r[n];
    for(int i=m-1..0) {
        s[n..0] = DgtSub(s[n..0], 1, y[n-1..0]);
        q[i] = q[i] + 1;
    } while(s[n]==0 & q[i]!=B-1);
    // restore s by adding y, if s became negative
    if(s[n]!=0) {
        s[n..0] = DgtSub(s[n..0], -1, y[n-1..0]);
        q[i] = q[i] - 1;
    }
    r[n-1..0] = s[n-1..0];
}
return (q,r);

Fig. B.11. Restoring Division of Radix-$B$ Numbers

by one. Now, observe that $x^{(i-1)} = v^{(i-1)} + y \cdot B^{i-1}$ holds, so that the next partial remainder $x^{(i-2)}$ can be obtained as follows:

\[
x^{(i-2)} = x^{(i-1)} - q_{i-2} \cdot y \cdot B^{i-2} = v^{(i-1)} + (B - q_{i-2}) \cdot y \cdot B^{i-2}
\]

Hence, instead of subtracting $y[n-1..0]$ from $s[n..0]$ in the first iteration for computing the next quotient digit (where we try $q_{i-2} = 1$), we add the scaled value $(B - 1) \cdot y$ as shown in Figure B.12. Note that this reintroduces the addition of scaled numbers, unless $B = 2$: In this case, the correction step becomes a simple addition in contrast to the simple subtractions (see Figure B.13).

A major problem of all of the previous algorithms for higher radices $B$ is that we have to try all possible digits in the worst case. As we will see below, there are good estimations of the quotient digits that can be used in all of the previous algorithms. However, the estimation we will obtain, requires that the most significant digit of the divisor is not zero, and therefore, we first have to show that this can be obtained in general as shown in the next lemma:

**Lemma B.16 (Dealing with Leading Zeros).** Assume the radix-$B$ numbers $x = \langle x_{m-1}, \ldots, x_0 \rangle_B^N$ and $y = \langle y_{n-1}, \ldots, y_0 \rangle_B^N$ with $y \neq 0$ are given for the division operation $(x \div y)$ and that $x$ and $y$ have some leading digits, i.e.,

- $x = \langle x_{m-1}, \ldots, x_k, x_{k-1}, \ldots, x_0 \rangle_B^N$ with $x_{m-1} = \ldots = x_k = 0$ and $x_{k-1} \neq 0$ for some $k \in \{1, \ldots, m\}$, and
function NonrestoreDivMod(digit x[m],y[n]) {
    digit s[n+1],q[m],r[n];
    for(int i=m-1..0) {
        // x(i+1) is stored in r[n-1..0]@x[i..0]
        if(s[n]!=0) // restore previous subtraction
            s[n..0] = DgtSub(r[n-1..0]@x[i],-(B-1),y[n-1..0]);
        else // no need to restore previous subtraction
            s[n..0] = DgtSub(r[n-1..0]@x[i],1,y[n-1..0]);
        q[i] = (s[n]==0)?1:0;
        while(s[n]==0 & q[i]!=B-1) {
            s[n..0] = DgtSub(s[n..0],1,y[n-1..0]);
            if(s[n]==0) q[i] = q[i] + 1;
        }
        r[n-1..0] = s[n-1..0];
    }
    if(s[n]!=0) {// restore remainder
        s[n..0] = DgtSub(s[n..0],-1,y[n-1..0]);
        r[n-1..0] = s[n-1..0];
    }
    return (q,r);
}

Fig. B.12. Nonrestoring Division of Radix-$B$ Numbers

• $y = \langle [y_{n-1},\ldots,y_\ell,\ldots,y_0]\rangle^N_B$ with $y_{n-1} = \ldots = y_\ell = 0$ and $y_{\ell-1} \neq 0$ for some $\ell \in \{1,\ldots,n\}$.

Then, we can skip the first $m - (k - \ell + 1)$ iterations to compute $x^{(i)}$ and instead, directly define $x^{(m)} = x^{(m-1)} = \ldots = x^{(k-\ell+1)} = x$ and $q_{m-1} = \ldots = q_{k-\ell+1} = 0$, and start the computation as described in the previous lemma with $x^{(k-\ell)}$ and $q_{k-\ell}$.

Proof. The maximal quotient is $(\text{MaxNat}_B(k) \text{ div } B^{\ell-1})$, i.e., by Lemma B.5 the number $\text{MaxNat}_B(k - \ell + 1)$. Therefore, the radix-$B$ representation of the quotient requires at most the $k - \ell + 1$ digits $q_{k-\ell},\ldots,q_0$. As we provide $m$ digits for the quotient, we can set the first $m - (k - \ell + 1)$ of them to zero. Since the digits $q_{m-1},\ldots,q_{k-\ell+1}$ are zero, we have $x^{(m)} = x^{(m-1)} = \ldots = x^{(k-\ell+1)} = x$.

If $x = \langle [x_{k-1},\ldots,x_0]\rangle^N_B$ and $y = \langle [y_{\ell-1},\ldots,y_0]\rangle^N_B$ with $x_{k-1} \neq 0$ and $y_{\ell-1} \neq 0$, the quotient $q = \langle [q_{k-\ell},\ldots,q_0]\rangle^N_B$ may have $k - \ell + 1$ digits. For the computation, we can start as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>x_{k-1}</th>
<th>x_{k-\ell}</th>
<th>x_{k-\ell-1}</th>
<th>\ldots</th>
<th>x_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_{m-n}</td>
<td>\ldots</td>
<td>0</td>
<td>y_{k-1}</td>
<td>y_{\ell}</td>
<td>y_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
function NonrestoreDivMod(digit x[m],y[n]) {
    digit s[n+1],q[m],r[n];
    for(int i=m-1..0) {
        // x(i+1) is stored in r[n-1..0]@x[i..0]
        if(s[n]!=0) // adding y
            s[n..0] = DgtSub(r[n-1..0]@x[i],-1,y[n-1..0]);
        else // subtracting y
            s[n..0] = DgtSub(r[n-1..0]@x[i],1,y[n-1..0]);
        q[i] = (s[n]==0)?1:0;
        r[n-1..0] = s[n-1..0];
    }
    if(s[n]!=0) {// restore remainder
        s[n..0] = DgtSub(s[n..0],-1,y[n-1..0]);
        r[n-1..0] = s[n-1..0];
    }
    return (q,r);
}

Fig. B.13. Nonrestoring Division of Radix-2 Numbers

and the computation proceeds as follows, where \(x_{i+1}^{(i)}\) may be zero or not (but definitely, we have \(y \cdot B < \langle x_{i+1}^{(i)}, \ldots, x_i^{(i)} \rangle_B\) according to \(I_2\)):

<table>
<thead>
<tr>
<th>0</th>
<th>...</th>
<th>0</th>
<th>(x_{i+1}^{(i)})</th>
<th>(x_{i+1}^{(i)})</th>
<th>(x_i^{(i+1)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_i)</td>
<td>...</td>
<td>0</td>
<td>(y_{i-1})</td>
<td>(y_{i-1})</td>
<td>(y_0)</td>
</tr>
</tbody>
</table>

In a software implementation, we can simply eliminate leading zeros before the division operation, while in a hardware implementation, the number of digits is usually fixed, so that we can not eliminate leading zeros. Nevertheless, the improvement considered in the above lemma is still important even for software realizations, since the second effect is that the lemma allows us to estimate how many quotient digits are zero, so that less iterations are required.

Another problem is to find the maximal quotient digit \(q_{i-1}\) so that \(q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)}\). Clearly, we could simply test all digits \(B-1, B-2, \ldots, 0\) (in this order) one after the other for \(q_{i-1}\) and stop as soon as \(q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)}\) holds (this is done in the algorithm of Figure B.9). For small \(B\), this is not too bad, but for large numbers \(B\) this may make the algorithm very slow. In particular, if \(B\) is chosen so that the digits correspond with the register length of a microprocessor, then \(B\) is too large for such a procedure.

A significant part of the research on efficient implementations of division algorithms, regardless on whether these are done in hardware or software therefore deals with the estimation of suitable quotient digits. To this end, the following lemma [146] is very useful:
Lemma B.17 (Estimating the Quotient Digit). Given the radix-$B$ numbers $x = (x_{\ell}, \ldots, x_0)_B$ and $y = (y_{\ell-1}, \ldots, y_0)_B$ with $y_{\ell-1} \neq 0$. Then, the maximal digit $q$ that satisfies $q \cdot y \leq x$ can be estimated as follows:

$$
\frac{x_{\ell} \cdot B + x_{\ell-1}}{y_{\ell-1} + 1} - 1 < q \leq \hat{q} := \min \left\{ \left\lfloor \frac{x_{\ell} \cdot B + x_{\ell-1}}{y_{\ell-1}} \right\rfloor, B - 1 \right\}
$$

Proof. We first prove the upper bound: It is clear that $q \leq \hat{q}$ holds if $\hat{q} = B - 1$, since $q \leq B - 1$ holds. Therefore, it remains to prove that $q \leq \hat{q}$ holds if $\hat{q} = ((x_{\ell} \cdot B + x_{\ell-1}) \text{ div } y_{\ell-1})$. By the division theorem, there is a number $r$ such that (1) $x_{\ell} \cdot B + x_{\ell-1} = \hat{q} \cdot y_{\ell-1} + r$ with (2) $0 \leq r < y_{\ell-1}$. Hence, we have (3) $\hat{q} \cdot y_{\ell-1} + r \geq x_{\ell} \cdot B + x_{\ell-1} - (y_{\ell-1} - 1)$, and therefore we have the following:

$$
x - \hat{q} \cdot y = x - \hat{q} \cdot (y_{\ell-1} \cdot B^{\ell-1} + \sum_{i=0}^{\ell-2} y_i \cdot B^i)
\leq x - \hat{q} \cdot y_{\ell-1} \cdot B^{\ell-1}
\leq x - (x_{\ell} \cdot B + x_{\ell-1} - (y_{\ell-1} - 1)) \cdot B^{\ell-1}
= x - x_{\ell} \cdot B^{\ell} - x_{\ell-1} \cdot B^{\ell-1} + (y_{\ell-1} - 1) \cdot B^{\ell-1}
= \left(\sum_{i=0}^{\ell-2} x_i \cdot B^i\right) + (y_{\ell-1} - 1) \cdot B^{\ell-1}
\leq (B^{\ell-1} - 1) + (y_{\ell-1} - 1) \cdot B^{\ell-1}
< B^{\ell-1} + (y_{\ell-1} - 1) \cdot B^{\ell-1}
= y_{\ell-1} \cdot B^{\ell-1}
\leq y
$$

Since $x - \hat{q} \cdot y \leq y$ is equivalent to $x \leq (\hat{q} + 1) \cdot y$, it follows that the maximal digit $q$ that satisfies $q \cdot y \leq x$ must be smaller than $\hat{q} + 1$, i.e., at most $\hat{q}$.

It remains to prove the lower bound. To this end, note that for the maximal digit $q$, we have the following inequations (4) $q \cdot y \leq x < (1 + q) \cdot y$, and therefore also (5) $q \leq \frac{x}{y} < 1 + q$. We derive a rougher estimation on the leading digits of $x$ and $y$ as follows.

$$
q \leq \frac{x_{\ell} \cdot B + x_{\ell-1} + B^{1-\ell} \cdot \sum_{j=0}^{\ell-2} x_j \cdot B^j}{y_{\ell-1} + B^{1-\ell} \cdot \sum_{j=0}^{\ell-2} y_j \cdot B^j} < 1 + q.
$$

Since, $0 \leq \sum_{j=0}^{\ell-2} x_j \cdot B^j < B^{\ell-1} - 1$ holds, we have $0 \leq B^{1-\ell} \cdot \sum_{j=0}^{\ell-2} x_j \cdot B^j < 1$, and therefore we obtain the following rougher estimations:

$$
(6) \quad q < \frac{x_{\ell} \cdot B + x_{\ell-1} + 1}{y_{\ell-1}} \quad \text{and} \quad (7) \quad \frac{x_{\ell} \cdot B + x_{\ell-1}}{y_{\ell-1} + 1} < 1 + q.
$$

Inequation (7) is the proposed lower bound and (6) is a weaker upper bound than $\hat{q}$ that we already proved.

The above estimations of the quotient digit can be quite sharp, but may in some cases be almost useless. Table B.1 shows the estimations in case $x_{\ell} = 0$. 

\Box
which is of special interest, since the estimation becomes then independent of \( B \), provided that we do not take into account that \( q \leq B - 1 \) holds for the upper bound. It is easily seen that the estimation is the better, the larger \( y_{\ell - 1} \) is. Knuth [146] proved the following interesting result on the upper bound \( \hat{q} \) that shows that an error \( \hat{q} - q \) greater or equal to 3 can only be obtained by digits \( 2 \cdot y_{\ell - 1} < B - 1 \):

**Lemma B.18 (Estimation the Quotient Digit).** Given the radix-\( B \) numbers \( x = \langle [x_{\ell}, \ldots, x_0] \rangle_B \) and \( y = \langle [y_{\ell - 1}, \ldots, y_0] \rangle_B \) with \( y_{\ell - 1} \neq 0 \). Let \( q \) be the maximal digit that satisfies \( q \cdot y \leq x \) and define \( \hat{q} \) as in Lemma B.17. Then, \( \hat{q} - q \geq 3 \) implies \( 2 \cdot y_{\ell - 1} < B - 1 \).

**Proof.** We already know by Lemma B.17 that \( q \leq \hat{q} \) holds, so that \( \hat{q} - q \geq 0 \). Now assume that (1) \( \hat{q} - q \geq 3 \), i.e., the error of our estimation is larger than 2. We obtain the inequation (2) \( \hat{q} \leq \frac{x_{\ell} \cdot B + x_{\ell - 1}}{y_{\ell - 1}} \) as follows:

\[
\hat{q} \leq \frac{x_{\ell} \cdot B + x_{\ell - 1}}{y_{\ell - 1}} = \frac{x_{\ell} \cdot B^\ell + x_{\ell - 1} \cdot B^{\ell - 1}}{y_{\ell - 1} \cdot B^{\ell - 1}} \leq \frac{x}{y_{\ell - 1} \cdot B^{\ell - 1}}
\]

Since \( y = y_{\ell - 1} \cdot B^{\ell - 1} + \sum_{i=0}^{\ell - 2} y_i \cdot B^i \), it follows that \( y < (y_{\ell - 1} + 1) \cdot B^{\ell - 1} \), and therefore (3) \( y - B^{\ell - 1} < y_{\ell - 1} \cdot B^{\ell - 1} \). By (2) and (3), we therefore conclude (4) \( \hat{q} < \frac{x}{y - B^{\ell - 1}} \). Note that \( y = B^{\ell - 1} \) (which would lead to a division by zero in (4)) is not possible, since this implies \( q = \hat{q} \) which however contradicts (1).

Since \( q \) is the maximal digit with \( q \cdot y \leq x \), we have \((q + 1) \cdot y > x \), which is equivalent to (5) \( q > \frac{x}{y} - 1 \). By (4) and (5), we now derive the following inequation:

\[
(6) \quad \hat{q} - q < \frac{x}{y - B^{\ell - 1}} - \left( \frac{x}{y} - 1 \right) = \frac{x}{y} \cdot \frac{B^{\ell - 1}}{y - B^{\ell - 1}} + 1
\]

Combining (6) and (1), we now obtain by transitivity of <
Lemma B.19 (Scaled Division). Given natural numbers \( x, y \neq 0 \), we have \((x \div y) = ((\lambda \cdot x) \div (\lambda \cdot y)) \) and \((x \mod y) = \lambda \cdot (x \mod y)\).

Proof. For \( q = (x \div y) \) and \( r = (x \mod y) \), we have \( x = q \cdot y + r \) with \( 0 \leq r < y \). Therefore, we obtain by multiplication with \( \lambda \) the equation \( \lambda \cdot x = q \cdot (\lambda \cdot y) + \lambda \cdot r \) with \( 0 \leq \lambda \cdot r < \lambda \cdot y \).

By the above lemma, we can multiply both the divisor and the dividend by a factor \( \lambda \neq 0 \) without changing the quotient. It is not too problematic that the remainder is changed, since we can obtain the remainder either as \( r := x - q \cdot y \) after we have determined the correct quotient \( q \) or even simpler by dividing by \( \lambda \).

The combination of the previous lemmata now reveals a very important improvement for our division algorithm:

Theorem B.20. Given the radix-\( B \) numbers \( x = ([x_\ell, \ldots, x_0])_B \) and \( y = ([y_\ell, \ldots, y_0])_B \) with \( y_\ell - 1 \neq 0 \) that occur in the division algorithm as operands. Define the scaling factor \( \lambda := \frac{B}{y_\ell + 1} \) to compute the new operands \( x' := \lambda \cdot x \) and \( y' := \lambda \cdot y \). Then, the number \( y' \) can be represented with the same number of digits as \( y \), and the most significant digit of \( y' \) has the property \( 2 \cdot y_\ell - 1 \geq B - 1 \), so that it follows for the quotient digit \( q \) and the estimation \( \hat{q} \) as described in Lemma B.17 that \( q \in \{\hat{q} - 2, \hat{q} - 1, \hat{q}\} \) holds.

Proof. Since \( x \) and \( y \) are numbers that occur in the division algorithm, we have \( x < y \cdot B \). If this would not be the case, then the previous quotient digit would be wrong since at least \( y \cdot B \) could have been additionally subtracted from \( x \). For this reason, we can see that the number of digits of \( y \) does not change by scaling with \( \lambda \) due to the following estimation:

\[
\frac{B}{y_\ell + 1} \cdot y < \frac{B}{y_\ell + 1} \cdot (y_\ell - 1) \cdot B^{\ell - 1} = B^\ell
\]
Note that the scaled $x$ may obtain one further digit. For the leading digit $y'_{\ell-1}$ of $y'$, we now have (with the carry $c$ from the lower significant digits):

$$y'_{\ell-1} := \lambda \cdot y_{\ell-1} + c = \frac{B}{y_{\ell-1} + 1} \cdot y_{\ell-1} + c \geq \frac{B}{2} + c$$

Note that $f(x) := \frac{x}{x+1}$ is monotonically increasing, and that therefore $\frac{1}{2} \leq \frac{y_{\ell-1}}{y_{\ell-1}+1}$ holds. Therefore, it follows by Lemma B.18 that the estimation $\hat{q} - 2$ is a lower bound for $q$ and by Lemma B.17 that $\hat{q}$ is an upper bound for the correct quotient digit $q$.

Our improved algorithm is shown in Figures B.14 and B.15. It improves our previous division algorithm so that in the worst case only three digits have to be tried to determine the correct quotient digit $q$ instead of trying all possible digits (which would be very bad for large $B$). The division algorithm of Figure B.14 first ‘eliminates’ leading zero digits, and then checks whether some simple cases occur. The first case is that the divisor has less digits than the dividend, and therefore $x < y$ holds (since we no longer have leading zero digits). Hence, we have $(x \ div \ y) = 0$ and $(x \ mod \ y) = x$. The next simple case is given when the divisor $y$ has only a single digit. In this case, we can use the more efficient division algorithm $\text{DivByDigit}$ shown in Figure B.14. Note that in this case, there is no need to estimate the quotient digit, we can easily compute it.

If these two simple cases are not given, then we have to perform a long division. To be able to compute tight estimations for the quotient digits, we therefore scale the divisor and the operand by the factor mentioned in the previous theorem, i.e., by $\lambda := \frac{B}{y^{(n-1)+1}}$. Then the scaled division algorithm is performed and the remainder is rescaled by dividing it by $\lambda$ with a call to the simple division algorithm $\text{DivByDigit}$. Note that scaling $y$ does not increase the number of digits. Hence, the final value of the variable $r_m$ after the loop in function $\text{Scale}$ when called for scaling $y$ will be zero, and therefore the index overflow that would occur by executing the then-branch of the final if-statement does never occur.

The core of the division algorithm is presented in Figure B.15, where we can already exclude the simple cases mentioned above. Hence, we have $2 \leq n \leq m$, no leading zero digits, and due to the scaling the property that the leading digit of $y$ is at least $\frac{2}{B}$. Hence, the division is now performed by estimating the quotient digit and then by trying one of the three possible values. Note that a lower bound is not required for the estimation, since the do-while loop stops as soon as a positive result of the subtraction is obtained. In most cases, this happens with the first attempt, and very few cases require all three attempts. Knuth presents some statistical arguments on the distribution of digits to support this proposition [146, 215]. There is even a better estimation of the quotient digit that reduces the error to 1. See also [101, 146] for an implementation that incorporates this additional issue.
function DivByDigit(int x[m], y) {
    int r, q[m];
    r = 0;
    for(int i=m-1..0) {
        r = r * B + x[i];
        q[i] = r div y;
        r = r mod y;
    }
    return (q, NewArray(1,r));
}

function Scale(int xs[m], x[n], lambda) {
    //assuming that n<m and 0<=lambda< B
    int rm=0;
    for(int i=0..n-1) {
        rm = lambda * x[i] + rm;
        xs[i] = rm mod B;
        rm = rm div B;
    }
    if(rm!=0) xs[n]=rm;
    return ();
}

function NatDivMod(int x[m], y[n]) {
    int k,l, lambda;
    int* xs,ys,q,r;
    // skip leading zero digits
    k=m-1; while(x[k]==0 & k>0) k=k-1;
    l=n-1; while(y[l]==0 & l>0) l=l-1;
    m=k+1; n=l+1;
    // simple case: y>x holds
    if(m<n) return (NewArray(1,0),x);
    // simple case: y has only one digit
    if(n==1) return DivByDigit(x,y[0]);
    // difficult case: prepare long division by prescaling
    lambda = B div (y[n-1]+1);
    xs = NewArray(m+1,0);
    Scale(xs,x,lambda);
    ys = NewArray(n,0);
    Scale(ys,y,lambda);
    (q,r) = ScaledDivision(xs,ys);
    (r,_r) = DivByDigit(r,lambda);
    return (q,r);
}

Fig. B.14. Improved Version of Division of Radix-$B$ Numbers (Part I)
function ScaledDivision(int x[m], y[n]) {
    //assuming 2<=n<=m and x[m-1]!=0 and y[n-1]!=0
    //and y[n-1]>=B div 2) thus allowing tight quotient bit estimation
    int qd,j,cs,sm,q[m-n+1],s[n];
    for(int i=m-n..0) {
        // estimate quotient, q[i] is then one of qd-2,qd-1,qd
        j = i + n;
        if(j==m)
            qd = (x[j-1]) div y[n-1];
        else
            qd = (x[j] * B + x[j-1]) div y[n-1];
        if(qd>=B) qd=B-1;
        // try digits q[i]=qd,qd-1,qd-2 until s<0
        qd = qd + 1;
        do {
            qd = qd-1;
            // compute s[n..0] = x[i+n..i]-qd * y[n-1..0]
            cs = 0;
            for(int k=0..n-1) {
                sm = cs + x[i+k] - qd * y[k];
                s[k] = sm mod B;
                cs = sm div B;
            }
            sm = cs + x[i+n];
            s[n] = sm mod B;
        } while(s[n]>0);
        // finally, we have q[i] and move s to x
        q[i] = qd;
        for(int k=0..n)
            x[i+k] = s[k];
    }
    return (q,r);
}

Fig. B.15. Improved Version of Division of Radix-$B$ Numbers (Part II)

There are many papers on the determination or the estimation of the quotient digit. The estimations we have presented so far are due to Knuth [146]. Software implementations often use radices $B$ that correspond with the word length of the processor, and therefore the improvement from $B$ guesses to at most 3 is a major improvement for the division algorithm. Hardware implementations are often use radices 4, 8, or 16 and do therefore also benefit from a good estimation. However, this is still not sufficient for high performance computing: Most hardware implementations of division do not make use of guesses at all. This is achieved by maintaining tables on the chip so that the quotient digit can be determined by some leading bits of $x$ and $y$. 
Estimations of the quotient digit for $B = 8$ Computer Arithmetic

<table>
<thead>
<tr>
<th>$(x_\ell, x_{\ell-1})$</th>
<th>$y_{\ell-1}$</th>
<th>$(x_\ell, x_{\ell-1})$</th>
<th>$y_{\ell-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 1)</td>
<td>0</td>
<td>(4, 1)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>0</td>
<td>(4, 2)</td>
<td>5, 6</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>0</td>
<td>(4, 3)</td>
<td>5</td>
</tr>
<tr>
<td>(0, 4)</td>
<td>0</td>
<td>(4, 4)</td>
<td>5, 6</td>
</tr>
<tr>
<td>(0, 5)</td>
<td>1</td>
<td>(4, 5)</td>
<td>5, 6</td>
</tr>
<tr>
<td>(0, 6)</td>
<td>1</td>
<td>(4, 6)</td>
<td>5, 6</td>
</tr>
<tr>
<td>(0, 7)</td>
<td>1</td>
<td>(4, 7)</td>
<td>5, 6</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>1</td>
<td>(5, 1)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>2</td>
<td>(5, 2)</td>
<td>6</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>2</td>
<td>(5, 3)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(1, 4)</td>
<td>2</td>
<td>(5, 4)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(1, 5)</td>
<td>2</td>
<td>(5, 5)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(1, 6)</td>
<td>2</td>
<td>(5, 6)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(1, 7)</td>
<td>2</td>
<td>(5, 7)</td>
<td>6, 7</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>3</td>
<td>(6, 1)</td>
<td>7</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>3</td>
<td>(6, 2)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>3</td>
<td>(6, 3)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(2, 4)</td>
<td>4</td>
<td>(6, 4)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(2, 5)</td>
<td>4</td>
<td>(6, 5)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(2, 6)</td>
<td>4</td>
<td>(6, 6)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(2, 7)</td>
<td>4</td>
<td>(6, 7)</td>
<td>7, 6</td>
</tr>
<tr>
<td>(3, 1)</td>
<td>4</td>
<td>(7, 1)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 2)</td>
<td>5</td>
<td>(7, 2)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 3)</td>
<td>5</td>
<td>(7, 3)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 4)</td>
<td>5</td>
<td>(7, 4)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 5)</td>
<td>5</td>
<td>(7, 5)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 6)</td>
<td>5</td>
<td>(7, 6)</td>
<td>7</td>
</tr>
<tr>
<td>(3, 7)</td>
<td>5</td>
<td>(7, 7)</td>
<td>7</td>
</tr>
</tbody>
</table>

Table B.2. Estimations of the quotient digit for $B = 8$ by the leading digits of $x$ and $y$.

The estimations we have considered in this section are not sufficient for this purpose: For example, Table B.2 shows that we would obtain by our estimations for radix $B = 8$ where only the prescaled operands are listed (note that entries with '*' can not occur due to the limited size of $x$). As can be seen, there are many cases where the digit is not determined, and there is even one case where three values are possible. Atkins [19, 20] gives a detailed analysis on the number of bits required to compute the quotient digit, see also [98, 100, 151, 185, 186, 195, 234, 264] for such algorithms. Finally, it should be remarked that the famous Pentium bug was due to an error in that table [93]. The look-up table for the quotient digit of the Pentium contains 1066 entries where in five cases the digit 2 has been erroneously replaced with 0.

Moreover, the most popular division algorithm for hardware implementations today, the SRT-division [185, 186], has been independently developed by Sweeney, Robertson [217], Tocher [?] which has been first observed by [102] who coined the name ‘SRT-division’ after the initials of the three inventors. The Pentium makes use of a radix-4 SRT-division algorithm with signed-digit numbers.

As can be seen, division is considerably more difficult than the other basic arithmetic operations. For this reason, many processor architectures omitted the division operation in the past and even today, some VLIW processors only provide means to efficiently implement the division algorithm in software.
B.3 The Integers

In the previous two sections, we have given a formalization of the natural numbers and their radix-$B$ representations together with detailed descriptions of algorithms for the basic arithmetic operations on the radix-$B$ representation. In the following two sections, we extend these algorithms to deal also with negative numbers. Analogous to the previous sections, we first discuss what we mean with integers, which is the purpose of this section. For the natural numbers, we used Peano’s axioms to determine the set of natural numbers up to isomorphism. For the integers, the following variant of Peano’s axioms can be used for the same purpose:

**Definition B.21 (Integers).** The set of integers $\mathbb{Z}$ contains the element $0$ and provides two functions $\text{succ}$ and $\text{pred}$ that satisfy the following axioms:

**Axiom 1a:** $\forall m \in \mathbb{Z}. \text{succ}(\text{pred}(n)) = n$

**Axiom 1b:** $\forall m \in \mathbb{Z}. \text{pred}(\text{succ}(n)) = n$

**Axiom 2a:** $\forall m,n \in \mathbb{Z}. \text{succ}(m) = \text{succ}(n) \rightarrow m = n$

**Axiom 2b:** $\forall m,n \in \mathbb{Z}. \text{pred}(m) = \text{pred}(n) \rightarrow m = n$

**Axiom 3:** $\forall P. \left( P(0) \land \left( \forall n \in \mathbb{N}. P(n) \rightarrow P(\text{succ}(n)) \land P(\text{pred}(n)) \right) \rightarrow \forall n \in \mathbb{N}. P(n) \right)$

We can prove that all integers can be written as either $\text{succ}^n(0)$ or as $\text{pred}^n(0)$, which gives us an analogous formalizations as for the natural numbers. Therefore, we can already conclude that the natural numbers are included in the set of the integers.

The next step is to also define arithmetic operations as shown in Figure B.16. Clearly, we follow the induction principle that is again given by the last axiom. The definitions are different to the operations that we used for the natural numbers in that we additionally have to define the cases for the predecessor terms, i.e., the negative numbers. Moreover, the subtraction operation is now fully defined, and we have added the absolute value as a new operation. It is easily seen that the operations we have defined for the natural numbers do match with the corresponding operators of the integers when we stay in the domain of the natural numbers.

We can prove again the laws given in Figure B.2. Additionally, we can add the following law $\forall a. \exists b. a + b = 0$, which states the existence of additive inverse elements for each integer. Hence, the set of integers is a ring, yet even an integral domain, and by the metric $\delta$ even an Euclidean ring. For this reason, the fundamental division theorem of Euclidean rings holds that we have already used for the natural numbers as well:

**Theorem B.22 (Division Theorem for Integers).** For all numbers $m, n \in \mathbb{Z}$ with $n \neq 0$, there are uniquely determined numbers $q$ and $r$ such that

1. $m = q \cdot n + r$
39° B Computer Arithmetic

- membership in \( \mathbb{N} \)
  - \( \text{IsNat}(0) : \Rightarrow \text{true} \)
  - \( \text{IsNat}(\text{succ}(n)) : \iff \text{succ}(n) = 0 \lor \text{IsNat}(n) \)
  - \( \text{IsNat}(\text{pred}(n)) : \iff \exists m \in \mathbb{Z}. n = \text{succ}(m) \land \text{IsNat}(m) \)

- comparisons
  - \( n \leq m : \iff \exists p. \text{IsNat}(p) \land n + p = m \)
  - \( n < m : \iff \neg (m \leq n) \)
  - \( n > m : \iff m < n \)
  - \( n \geq m : \iff m \leq n \)

- addition
  - \( m + 0 = m \)
  - \( m + \text{succ}(n) = \text{succ}(m + n) \)
  - \( m + \text{pred}(n) = \text{pred}(m + n) \)

- subtraction
  - \( m - 0 = 0 \)
  - \( m - \text{succ}(n) = \text{pred}(m - n) \)
  - \( m - \text{pred}(n) = \text{succ}(m - n) \)

- multiplication
  - \( m \cdot 0 = 0 \)
  - \( m \cdot \text{succ}(n) = m \cdot n + m \)
  - \( m \cdot \text{pred}(n) = m \cdot n - m \)

- modulo operation
  - \( (0 \mod m) = 0 \)
  - \( (\text{succ}(n) \mod m) = \begin{cases} 0 & : \text{if } \text{succ}(n \mod m) = m \\ \text{succ}(n \mod m) & : \text{otherwise} \end{cases} \)
  - \( (\text{pred}(n) \mod m) = \begin{cases} \text{pred}(n \mod m) & : \text{if } (n \mod m) = 0 \\ \text{pred}(n \mod m) & : \text{otherwise} \end{cases} \)

- division
  - \( (0 \div m) = 0 \)
  - \( (\text{succ}(n) \div m) = \begin{cases} \text{succ}(n \div m) & : \text{if } \text{succ}(n \mod m) = m \\ (n \div m) & : \text{otherwise} \end{cases} \)
  - \( (\text{pred}(n) \div m) = \begin{cases} \text{pred}(n \div m) & : \text{if } (n \mod m) = 0 \\ (n \div m) & : \text{otherwise} \end{cases} \)

- distance
  - \( \delta(m, n) := \begin{cases} m - n & : \text{if } n \leq m \\ n - m & : \text{otherwise} \end{cases} \)

- absolute value
  - \( |m| := \begin{cases} m & : \text{if } 0 \leq m \\ 0 - m & : \text{otherwise} \end{cases} \)

Fig. B.16. Primitive Recursive Definitions of Arithmetic Operators for the Integers

2. \( 0 \leq r < |n| \)

Moreover, we have \( q = (m \div n) \) and \( r = (m \mod n) \).

Proof. The proof we have given for Theorem B.2 in case of the natural numbers can be repeated here. We only have to add a case for the predecessors, which can be analogously proved.
Note that according to the above theorem, \((m \mod n)\) is always a natural number. Unfortunately, the above mathematical definition of \(q = (m \div n)\) and \(r = (m \mod n)\) is not always used in programming languages and hardware implementations. In some cases, the quotient is rounded towards zero or towards \(-\infty\) which yields in both cases a different definition of \(\text{div}\) and \(\text{mod}\).

## B.4 B-Complement Numbers

Having formalized the set of integers, we are now interested in adapting the radix-\(B\) representation of the natural numbers to also deal with the integers. To this end, there are at least the following choices:

- We use a boolean value to store the sign of the number, and store its absolute value in form of an ordinary radix-\(B\) number.
- For fixed numbers of digits, we use a bias \(b\) that is subtracted from the unsigned value of the radix-\(B\) number, i.e., \(\langle x \rangle^N_B - b\).
- We use \(B\)-complement numbers as defined below.

Each of these number representations has advantages and disadvantages. The use of an explicit sign is attractive since we could directly use the algorithms for radix-\(B\) arithmetic operations. However, the explicit sign representation is seldom used, in particular, since the zero (a frequent number) has two representations. The use of a bias is very inconvenient: The bias and therefore, the numbers and the algorithms depend on the chosen number of digits which can not be simply extended. In particular, multiplication and division become really difficult with a bias. Hence, the most advantages are found in the \(B\)-complement representation as defined below:

**Definition B.23 (B-Complement Radix Numbers).** Given a natural number \(B > 1\) called the radix, every list \([x_{n-1}, \ldots, x_0]\) of digits is mapped to an integer as follows:

\[
\langle [x_{n-1}, \ldots, x_0] \rangle^Z_B := -x_{n-1} \cdot B^{n-1} + \sum_{i=0}^{n-2} x_i \cdot B^i
\]

Moreover, the integer \(\langle [x_{n-1}, \ldots, x_0] \rangle^Z_B\) is a \(B\)-complement number if the entries \(x_i\) are digits of the radix \(B\), i.e., \(x_i \in \{0, \ldots, B-1\}\).

As can be seen, \(B\)-complement numbers are closely related with radix-\(B\) numbers, and often \(B\)-complement numbers are also called radix-\(B\) numbers, since they are also based on a radix \(B\) and a positional number system where the positions of the digits are assigned a different weight. The only difference between radix-\(B\) numbers and the \(B\)-complement numbers is that the most significant digit of a \(B\)-complement number is counted negatively. For
this reason, we have the following relationship that we will often use to lift theorems from radix-\( B \) numbers to \( B \)-complement numbers:

\[
\langle [x_{n-1}, \ldots, x_0] \rangle_B^Z = -x_{n-1} \cdot B^{n-1} + \langle [x_{n-2}, \ldots, x_0] \rangle_B^N
\]

Another simple relationship between \( B \)-complement and radix-\( B \) numbers is the following equation:

\[
\langle [x_{n-1}, \ldots, x_0] \rangle_B^Z = \langle [-x_{n-1}, x_{n-2}, \ldots, x_0] \rangle_B^N
\]

For fixed numbers of digits, we can easily derive the following bounds on \( B \)-complement numbers with \( n \) digits:

**Lemma B.24 (Fixed Number of Digits).** The set of integers that is represented by \( B \)-complement \( n \)-digit numbers is \( \{ \text{MinInt}(n), \ldots, \text{MaxInt}(n) \} \), where

- \( \text{MaxInt}(n) = \langle [0, B-1, \ldots, B-1] \rangle_B^Z = B^{n-1} - 1 \) \( n \) digits
- \( \text{MinInt}(n) = \langle [B-1, 0, \ldots, 0] \rangle_B^Z = -B^{n-1} \cdot (B-1) \) \( n \) digits

A first annoying fact can now be observed: there are more negative numbers than there are positive numbers. This means that if we change the sign, we may have to add further digits, or when working with fixed digit sets, it will not always be possible to change the sign.

By the above lemma, it is clear that every integer that can be represented with \( n \) digits can also be represented with \( n + m \) digits. For the natural numbers, we simply added leading zero digits, and we can still do this if the leading digit \( x_{n-1} \) is zero, i.e., when the considered integer is also a natural number. For negative numbers, it is also possible to add further digits without changing the represented value:

**Lemma B.25 (Digit Extension for \( B \)-Complement Numbers).** Given the integer \( x = \langle [x_{n-1}, \ldots, x_0] \rangle_B^Z \) with \( n \) digits, the uniquely defined representation with \( n + m \) digits is as follows:

\[
x = \begin{cases} 
\langle [0, \ldots, 0, x_{n-1}, \ldots, x_0] \rangle_B^Z & : \text{if } x_{n-1} = 0 \\
\langle [1, B-1, \ldots, B-1, B-x_{n-1}, x_{n-2}, \ldots, x_0] \rangle_B^Z & : \text{if } x_{n-1} \neq 0 
\end{cases}
\]

**Proof.** The case for \( x_{n-1} = 0 \) is clear, so we consider only the second case. It is sufficient to prove this case for one additional digit, since that procedure can then be repeated as often as desired. The proof is simply as follows:

\[
\langle [1, B-x_{n-1}, \ldots, x_0] \rangle_B^Z = -1 \cdot B^n + (B-x_{n-1}) \cdot B^{n-1} + \langle [x_{n-2}, \ldots, x_0] \rangle_B^N
\]

\[
= (-B + B-x_{n-1}) \cdot B^{n-1} + \langle [x_{n-2}, \ldots, x_0] \rangle_B^N
\]

\[
= -x_{n-1} \cdot B^{n-1} + \langle [x_{n-2}, \ldots, x_0] \rangle_B^N
\]

\[
= \langle [x_{n-1}, \ldots, x_0] \rangle_B^Z
\]
For the important case $B = 2$, the above lemma is often called the ‘sign extension’ lemma, since the leading digit, regardless whether it is 0 or 1, is repeated as often as desired. However, the name ‘sign extension’ is misleading, since the leading digit is not the sign as in an explicit sign representation even though the leading digit tells us whether the number is negative.

The digit extension lemma has also the remarkable consequence that we can represent every integer as a $B$-complement number that starts either with the digit 0 or 1 (independent of the radix $B$), and clearly, numbers starting with 0 are positive, while numbers starting with 1 are negative. Moreover, instead of extending a $B$-complement number by further digits, we may also use the lemma to remove redundant leading digits, since the following simple consequences hold:

- $\langle [0, 0, x_{n-1}, \ldots, x_0] \rangle_B^Z = \langle [0, x_{n-1}, \ldots, x_0] \rangle_B^Z$
- $\langle [1, x_{n-1}, \ldots, x_0] \rangle_B^Z = \langle [B - x_{n-1}, \ldots, x_0] \rangle_B^Z$ for $x_{n-1} \neq 0$

Note that a number has no redundant leading digits iff the leading digit $x_{n-1}$ is neither 0 nor 1 or otherwise, if the two leading digits $x_{n-1}, x_{n-2}$ are either $x_{n-1} = 0 \land 0 < x_{n-2}$ or $x_{n-1} = 1 \land 0 < x_{n-2}$.

This is rather different from the everyday use of integers where explicit sign representations are used: We are used to eliminate leading zeros, but for $B$-complement numbers this may modify the represented number in that a positive number will become negative. We should therefore write 03 instead of 3, since $\langle [0, 3] \rangle_{10}^Z = 3$, but $\langle [3] \rangle_{10}^Z = -3$.

For the conversion of a given integer to a $B$-complement number for some chosen radix $B$, we can use the following lemma:

**Lemma B.26 (Conversion to $B$-Complement).**

Given the $B$-complement number $x = \langle [x_{n-1}, \ldots, x_0] \rangle_B$ with $n$ digits, then the following holds for any $k \in \{0, \ldots, n-1\}$:

- $(x \mod B^k) = \langle [x_{k-1}, \ldots, x_0] \rangle_N^B$
- $(x \div B^k) = \langle [x_{n-1}, \ldots, x_k] \rangle_Z^B$

In particular, $x_k = ((x \div B^k) \mod B)$ for $k \in \{0, \ldots, n-2\}$ and $x_{n-1} = -((x \div B^{n-1})$. Moreover, the minimal number of digits sizeOf(x) required to represent the integer $x$ is computed as follows:

$$sizeOf(x) = \begin{cases} 
1 + \lceil \log_B(x + 1) \rceil : & \text{if } x > 0 \\
1 : & \text{if } x = 0 \\
1 + \lceil \log_B(-x) \rceil : & \text{if } x < 0 
\end{cases}$$

**Proof.** The proof is very simple: We obviously have the following equation:

$$x = \langle [x_{n-1}, \ldots, x_k] \rangle_B^Z \cdot B^k + \langle [x_{k-1}, \ldots, x_0] \rangle_N^B$$

Moreover, we have $0 \leq \langle [x_{k-1}, \ldots, x_0] \rangle_N^B \leq \text{MaxNat}_B(k) < B^k$, so that the rest follows by Theorem B.22.
Note that the above Lemma can be combined with Lemma B.25 to compute also values for division by $B^k$ for larger $k$. We can therefore compute the digits of a given number by successively dividing it by $B$ and taking the sequence of moduli as the digits $x_0, x_1, \ldots$ of $x$. Finally, we must obtain either $x = 0$ in case that the given number $x$ was nonnegative or $x = -1$ in case the number was negative (consider again Lemma B.25 to see this and note that $\langle[1, B-1, \ldots, B-1]\rangle_B = -1$).

By the above lemmata, we now conclude the following result that is essential for the use of the $B$-complement numbers:

**Theorem B.27 (Unique Representation).** Every integer $n$ can be represented in the $B$-complement number system with a minimal number of digits $\ell_n$ as specified in Lemma B.26. For every number of digits $\ell \geq \ell_n$, the $B$-complement representation of $n$ is uniquely determined.

```cpp
function BComplement(int x[n]) {
    int s, c=0, y[*];
    for(int i=0..n-1) {
        s = (i==n-1)?c-x[i]:c+x[i];
        y[i] = s mod B;
        c = s div B;
        i = i+1;
    }
    i = n;
    while(c!=0 & c!=-1) {
        y[i] = c mod B;
        c = c div B;
        i = i+1;
    }
    y[i] = -c;
    return y;
}
```

**Fig. B.17.** Conversion of $B$-Complement Numbers

Using the above results, we can convert any integer to its uniquely determined $B$-complement representation (up to leading redundant digits). The main advantage of $B$-complement numbers is that the algorithms of the arithmetic operations that we have already developed in the previous sections can almost be retained with certain changes for the most significant digits. In particular, the following equations are valid and almost lead to the first algorithms of arithmetic operations:

- $\langle[x_{n-1}, \ldots, x_0]\rangle_B + \langle[y_{n-1}, \ldots, y_0]\rangle_B = \langle[x_{n-1} + y_{n-1}, \ldots, x_0 + y_0]\rangle_B$
• \(\langle x_{n-1}, \ldots, x_0 \rangle_B^B = \langle y_{n-1}, \ldots, y_0 \rangle_B^N = \langle [x_{n-1} - y_{n-1}, \ldots, x_0 - y_0] \rangle_B^N\n
\langle x_{n-1}, \ldots, x_0 \rangle_B^B \cdot y = \langle [x_{n-1} \cdot y, \ldots, x_0 \cdot y] \rangle_B^N\n
Similar to radix-\(B\) numbers, the results may however not belong to the chosen digit set, and are therefore not yet \(B\)-complement numbers. However, a similar normalization procedure as discussed in Theorem B.8 can be used to correct this.

**Theorem B.28 (Conversion to \(B\)-Complement).**

Given an expression \(\langle x_{n-1}, \ldots, x_0 \rangle_B^B\), function \(\text{BComplement}\) shown in Figure B.17 can be used to compute an equivalent \(B\)-complement number.

The proof of the above theorem is very simple and follows essentially from the previous theorem, since the quotient and the remainder, and therefore the digits of a \(B\)-complement number are uniquely determined. Hence, we can simply adapt the addition and subtraction algorithms for radix-\(B\) numbers to obtain analog algorithms for \(B\)-complement numbers. However, before we consider the four basic arithmetic operations, let us first consider the comparison relation.

**Theorem B.29 (Comparison of \(B\)-Complement Numbers).** Given the \(B\)-complement numbers \(x = \langle [x_{n-1}, \ldots, x_0] \rangle_B^B\) and \(y = \langle [y_{n-1}, \ldots, y_0] \rangle_B^B\), the following holds:

• \(x = y \iff \forall i \in \{0, \ldots, n - 1\}. x_i = y_i\)

• \(x < y \iff \begin{cases} y_{n-1} < x_{n-1} & : \text{if } x_{n-1} \neq y_{n-1} \cr \langle [x_{n-2}, \ldots, x_0] \rangle_B^N < \langle [y_{n-2}, \ldots, y_0] \rangle_B^N & : \text{if } x_{n-1} = y_{n-1} \end{cases}\)

The result is clear for equality, since every number that can be represented with \(n\) digits has a unique representation. Therefore, we can simply test whether for all digits \(x_i\) and \(y_i\) of given numbers \(x\) and \(y\), we have \(x_i = y_i\). For testing \(x < y\), the above result can be seen as follows:

\[
y - x = \langle [y_{n-1}, \ldots, y_0] \rangle_B^B - \langle [x_{n-1}, \ldots, x_0] \rangle_B^B = (x_{n-1} - y_{n-1}) \cdot B^{n-1} + \langle [y_{n-2}, \ldots, y_0] \rangle_B^N - \langle [x_{n-2}, \ldots, x_0] \rangle_B^N\n\]

If \(x_{n-1} = y_{n-1}\) holds, \(y - x > 0\) is equivalent to \(\langle [y_{n-2}, \ldots, y_0] \rangle_B^N - \langle [x_{n-2}, \ldots, x_0] \rangle_B^N > 0\), which is a test for radix-\(B\) numbers. For \(x_{n-1} \neq y_{n-1}\), there is no need for a recursive call, since the result is already determined by the most significant digits, i.e., \(x < y\) iff \(y_{n-1} < x_{n-1}\). The reason for this is that the difference of the most significant digits can not be outweighed by the remaining digits: \(\langle [y_{n-2}, \ldots, y_0] \rangle_B^N - \langle [x_{n-2}, \ldots, x_0] \rangle_B^N \) is at most \(\text{MaxNat}_B(n - 2)\), i.e., \(B^n - 1\), while \(|x_{n-1} - y_{n-1}| \cdot B^{n-1} \geq B^{n-1}\).

Hence, we obtain the small algorithm shown in Figure B.18 for the comparison of \(B\)-complement numbers. Hence, hardware or software implementations for comparison can be used for radix-\(B\) and \(B\)-complement numbers can share a lot of code.
function IntLess(digit x[n],y[n]) {
    if(x[n-1]==y[n-1]) return NatLess(x,y);
    else return y[n-1]<x[n-1];
}

Fig. B.18. Comparison of B-Complement Numbers

For the construction of addition and subtraction algorithms, we can proceed in the same way as for radix-B numbers: We can first add the B-complement numbers digit by digit and use then the function BComplement of Figure B.17 to convert the resulting sum to a B-complement number. Analogously, a subtraction algorithm is obtained by first subtracting the digits of the second operand from the corresponding digits of the first operand, and a subsequent conversion to a B-complement number.

Clearly, we can also merge the two loops, i.e., the first loop for digitwise addition and subtraction with the second loop for B-complement conversion. The resulting algorithms are shown in Figure B.19 and their correctness is stated in the following theorem.

Theorem B.30 (Addition and Subtraction of B-Complement Numbers). Given numbers $x = \langle x_{n-1}, \ldots, x_0 \rangle_B$ and $y = \langle y_{n-1}, \ldots, y_0 \rangle_B$, and the results $(c_a, [a_{n-1}, \ldots, a_0])$ and $(c_b, [b_{n-1}, \ldots, b_0])$ of the function calls IntAdd$(x, y)$ and IntSub$(x, y)$ for the functions given in Figure B.19. Then, the following holds:

- $\langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B^Z$ is a B-complement number
- $x + y = \langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B^Z$
- $\langle [c_b, b_{n-1}, \ldots, b_0] \rangle_B^Z$ is a B-complement number
- $x - y = \langle [c_b, b_{n-1}, \ldots, b_0] \rangle_B^Z$

$\langle [c_a, a_{n-1}, \ldots, a_0] \rangle_B^Z$ can be represented with $n$ digits iff $c_a = a_{n-1} = 0$ or $c_a = 1$ and $a_{n-1} \neq 0$ (the analog statement holds for $\langle [c_b, b_{n-1}, \ldots, b_0] \rangle_B^Z$).

Proof. The above statements follow almost directly from Lemma B.28 and the preceding trivial digitwise addition and subtraction. The proposition on the representation of the sums with $n$ digits follows directly from Lemma B.25.

The subtraction algorithm gives us another algorithm for testing whether $x < y$ holds: after performing the subtraction operation we have $c_b = 0 \Leftrightarrow x < y$ holds for the result $(c_b, [b_{n-1}, \ldots, b_0])$.

Compared to the addition and subtraction of radix-B numbers, there are only modifications for the most significant digits. Therefore, we could alternatively use the radix-B algorithms except for the most significant digits and provide some extra code for the most significant digits. Moreover, it is not difficult to see that we could alternatively use the radix-B addition for adding $[-x_{n-1}, x_{n-2}, \ldots, x_0]$ and $[-y_{n-1}, y_{n-2}, \ldots, y_0]$ and changing
The sign of the carry digit. Analogously, we can use radix-$B$ subtraction of $[-x_{n-1}, x_{n-2}, \ldots, x_0]$ and $[-y_{n-1}, y_{n-2}, \ldots, y_0]$ with a subsequent change of the sign of the carry digit.

The subtraction $0 - x$ allow us furthermore to change the sign of a given $B$-complement number $x$, i.e., to compute the $B$-complement representation of $-x$. The same number $-x$ can alternatively be obtained by first changing the sign of the digits $-x = \langle -x_{n-1}, \ldots, -x_0 \rangle_2$ and converting this intermediate result via the normalization algorithm given in Figure B.17 to the allowed digit set. As an example, consider the number $\langle 4, 5 \rangle_{10}$ which is $-40 + 5 = -35$. We first compute $\langle -4, -5 \rangle_{10}$ and convert this intermediate result to the $B$-complement number $\langle 0, 3, 5 \rangle_{10}$.

One may wonder whether the $B$-complement that we defined for natural numbers could also be used for subtraction of $B$-complement numbers. However, this is only reasonable for $B = 2$: We may even think of a complement operation as follows: first change the digits $x_i$ to $B - 1 - x_i$ and then add some
constant value $\alpha$ to the obtain the complementary number. Then, we would obtain the following:

$$\langle [x_{n-1}, \ldots, x_0] \rangle_B + \langle [B - 1 - x_{n-1}, \ldots, B - 1 - x_0] \rangle_B + \alpha$$

$$= \langle [B - 1, \ldots, B - 1] \rangle_B + \alpha$$

$$= \langle [B - 1, \ldots, B - 1] \rangle_B + \alpha$$

$$= -(B - 1) \cdot B^{n-1} + (B^{n-1} - 1) + \alpha$$

$$= -(B - 1) \cdot B^{n-1} + (B^{n-1} - 1) + \alpha$$

$$= B^{n-1} \cdot (1 - (B - 1)) + (\alpha - 1)$$

$$= B^{n-1} \cdot (2 - B) + (\alpha - 1)$$

To obtain $-x$ by this procedure, the sum must become zero, which is the case for $\alpha = (B - 2) \cdot B^{n-1} + 1$. This holds independent of $n$ for $B = 2$ and $\alpha = 1$. For other values of $B$, $\alpha$ depends on $n$ so that the addition of the complement is not very useful.

The multiplication of $B$-complement numbers can be done by the same principle as for the natural numbers:

$$x \cdot y = x \cdot \left( -y_{n-1} \cdot B^{n-1} + \sum_{i=0}^{n-2} y_i \cdot B^i \right) = -y_{n-1} \cdot x \cdot B^{n-1} + \sum_{i=0}^{n-2} y_i \cdot x \cdot B^i$$

Hence, we simply have to add the partial products $y_i \cdot x \cdot B^i$ for $i = 0, \ldots, n - 2$ and finally the remaining partial product $-y_{n-1} \cdot x \cdot B^{n-1}$. Compared to the multiplication of radix-$B$ numbers, we therefore only have the following changes:

- The additions have to be done with the addition of $B$-complement numbers instead of the addition of radix-$B$ numbers which is different for the most significant sum digit and the additional carry digit.
- The last summand that has to be added is multiplied with $-y_{n-1}$ instead of $y_{n-1}$.

Except for these modifications, the algorithm is the same. The resulting algorithm is shown in Figure B.20 and the example multiplications $(-5) \cdot 3 = -15$ and $(-5) \cdot (-3) = 15$ for $B = 2$ are shown below:

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Hence, we have $\langle [1, 0, 1, 1] \rangle_2 \cdot \langle [0, 1, 1, 1] \rangle_2 = \langle [1, 1, 1, 1, 0, 0, 0, 1] \rangle_2$ and $\langle [1, 0, 1, 1] \rangle_2 \cdot \langle [1, 0, 1, 0, 0, 0, 0, 1] \rangle_2 = \langle [0, 0, 0, 1, 1, 1, 1, 1] \rangle_2$. 
function IntMult(int x[m], y[n]) {
    int p[m+n], d, s, c;
    for(int i=0..n-1) {
        // loop body assigns p[m+i..i] = p[m+i-1..i] + x[m-1..0] * y[i];
        c = 0;
        d = (i==n-1)?-y[i]:y[i];
        for(int j=0..m-2) {
            s = p[i+j] + x[j] * d + c;
            p[i+j] = s mod B;
            c = s div B;
        }
        s = c-(p[i+m-1] + x[m-1] * d);
        p[i+m-1] = s mod B;
        p[i+m] = -(s div B);
    }
    return p;
}

Fig. B.20. Multiplication of B-Complement Numbers

Finally, it remains to consider the division of B-complement numbers, which is again the most difficult operation of the arithmetic operators. The problem is here to determine for given B-complement numbers x and y the uniquely determined B-complement numbers q and r that satisfy the equation \( a = y \cdot q + r \) and the inequations \( 0 \leq r < |y| \). As we will see, a lot of the radix-B algorithms can be retained, although the B-complement division has many special problems.

We first have to determine the numbers of digits needed to represent the quotient and the remainder of a m-digit number x and a n digit number y. Clearly, the larger the absolute values of these numbers, the more digits are required. The maximal absolute value of the quotient \( \lfloor x \div y \rfloor \) is obtained by choosing the maximal absolute value of x and the minimal absolute value of y, i.e., \( x = \pm \text{MinInt}(m) \) and \( y = \pm 1 \). By checking the four cases, it turns out that the worst case example is \( \text{MinInt}(m) \div (-1) \) which is \(-\text{MinInt}(m)\) and which requires \( m + 1 \) digits. Since we have \( 0 \leq r < |y| \), it follows that the extremal case for the remainder is \( r = |y| - 1 \), which is maximal for \( |\text{MinInt}(m)| - 1 = \langle [0, B-2, B-1, \ldots, B-1] \rangle_B \). Hence, the remainder requires for \( B > 2 \) \( n + 1 \) digits, and only \( n \) digits in case \( B = 2 \). As the remainder is always non-negative, we could also return it as a radix-B number so that the leading digit could be saved. However, we will not do so in the following.

In the general case, we therefore have to compute the \( m + 1 \) digits \( q = \langle [q_m, \ldots, q_0] \rangle_B \) for the quotient, and if \( B > 2 \) holds, we have to compute the \( n + 1 \) digits \( r = \langle [r_n, \ldots, r_0] \rangle_B \) for the remainder while for \( B = 2 \) only the \( n \) digits \( r = \langle [r_{n-1}, \ldots, r_0] \rangle_2 \) are sufficient. Having determined the number of
required digits, we can now start with the computation of these digits. To this end, the simplest way to compute the desired values is to reduce the given $B$-complement division problem to a corresponding radix-$B$ division problem. The following theorem shows how this can be effectively done.

**Theorem B.31 (Reducing $B$-Complement to Radix-$B$ Division).** Given integers $x$ and $y$, we compute $(x \div y)$ and $(x \mod y)$ using the radix-$B$ division algorithm shown in Figures B.14 and B.15 as follows:

1. If $x < 0$ and $y > 0$, then we compute $q := ((-x) \div y)$ and $r := ((-x) \mod y)$ to obtain $(x \div y)$ and $(x \mod y)$ as follows:
   
   $$(x \div y) := \begin{cases} \frac{-q + 1}{r + 1} : & \text{if } r > 0 \\ \frac{-q}{r} : & \text{if } r = 0 \end{cases} \quad (x \mod y) := \begin{cases} y - r : & \text{if } r > 0 \\ r : & \text{if } r = 0 \end{cases}$$

2. If $x < 0$ and $y < 0$, then we compute $q := ((-x) \div (-y))$ and $r := ((-x) \mod (-y))$ to obtain $(x \div y)$ and $(x \mod y)$ as follows:
   
   $$(x \div y) := \begin{cases} q + 1 : & \text{if } r > 0 \\ q : & \text{if } r = 0 \end{cases} \quad (x \mod y) := \begin{cases} -y - r : & \text{if } r > 0 \\ r : & \text{if } r = 0 \end{cases}$$

3. If $x > 0$ and $y < 0$, then we compute $q := (x \div (-y))$ and $r := (x \mod (-y))$ to obtain $(x \div y)$ and $(x \mod y)$ as follows:
   
   $$(x \div y) := -q \quad (x \mod y) := r$$

**Proof.** If $x < 0$ and $y > 0$, then we have for $q := ((-x) \div y)$ and $r := ((-x) \mod y)$ the facts (1) $-x = q \cdot y + r$ and (2) $0 < r < |y| = y$. Multiplying (1) with $-1$ yields (3) $x = -(q) \cdot y - r$. For $r = 0$, we therefore already have what we need to define $(x \div y)$ and $(x \mod y)$. For $r > 0$, we first use (3) to obtain $x = -(q) \cdot y - r = -(q) \cdot y + y - y - r = -(q + 1) \cdot y + (y - r)$. Since $0 < r < y$ implies $y > y - r > 0$, the first proposition holds.

If $x < 0$ and $y < 0$, then we have for $q := ((-x) \div (-y))$ and $r := ((-x) \mod (-y))$ the facts (4) $-x = q \cdot (-y) + r$ and (5) $0 \leq r < |y| = y$. Multiplying (4) with $-1$ yields (6) $x = q \cdot y - r$. For $r = 0$, we therefore already have again what we need to define $(x \div y)$ and $(x \mod y)$. For $r > 0$, we transform (6) similarly as above $x = q \cdot y - r = q \cdot y + y - y - r = (q + 1) \cdot y + (-y - r)$. Moreover, multiplying (5) with $-1$ and considering that $r > 0$ holds, yields (7) $0 > -r > y$, and adding $-y$ yields (8) $-y > -y + y > 0$. Since, $|y| = -y$ holds, we therefore have $0 \leq -y - r < y = |y|$.

Finally, if $x > 0$ and $y < 0$, then we have for $q := (x \div (-y))$ and $r := (x \mod (-y))$ the facts (9) $x = q \cdot (-y) + r$ and (10) $0 \leq r < |y| = -y$. (9) yields directly (11) $x = -(q) \cdot y + r$, which completes the proof.

\[\square\]

Due to the above theorem, we can reduce the $B$-complement division to non-negative divisors and dividends. For this reason, we can use the radix-$B$ division algorithms of the previous chapter to perform the $B$-complement division with the corresponding reduction steps stated in the above theorem.
The above theorem is not at all surprising. We could present such a theorem also for addition, subtraction, and multiplication. However, for division, the reduction to the corresponding radix-$B$ algorithm is more interesting since the division of $B$-complement numbers is more complicated. For this reason, the trade-off to spend computation time on reduction to radix-$B$ case is reasonable.

The next theorem is the core of all subtractive $B$-complement division algorithms. However, the algorithm described in that theorem only computes a preliminary quotient that has negative digits in case of negative divisors:

**Theorem B.32 (Division of $B$-Complement Numbers).**

For $B$-complement numbers $x = \langle [x_{m-1}, \ldots, x_0] \rangle_B$ and $y = \langle [y_{n-1}, \ldots, y_0] \rangle_B$, we define the $m+1$ $B$-complement numbers $x^{(i)}$ and the $m+1$ digits $q_i$ recursively as follows:

- $q_m := \begin{cases} 1 & : \text{if } x < 0 \land y > 0 \\ 0 & : \text{if } x \geq 0 \\ -1 & : \text{if } x < 0 \land y < 0 \end{cases}$
- $x^{(m)} := x + q_m \cdot y \cdot B^m$, where $q_m$ is defined above.
- $x^{(i-1)} := x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$ for $i = m, \ldots, 1$, where $q_{i-1}$ is chosen from $\{-B^{-1}, \ldots, 0, \ldots, B-1\}$ such that (1) $|q_{i-1}|$ is maximal, (2) $q_{i-1} \cdot y \geq 0$, and that (3) $q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)}$ holds.

For all $i$, we then have the invariants ($I_1$) $x^{(i)} + \langle [q_m, \ldots, q_i] \rangle_B \cdot y \cdot B^i = x$ and ($I_2$) $0 \leq x^{(i)} < |y| \cdot B^i$. Therefore, the remainder $r$ is the $B$-complement number $r = x^{(0)}$, while the quotient $q$ can be obtained by $B$-complement conversion (Figure B.17) of the number $q = \langle [q_m, \ldots, q_0] \rangle_B$.

**Proof.** We first prove invariant ($I_1$): For $i = m$, we have $x^{(m)} = x + q_m \cdot y \cdot B^m = x - \langle [q_m] \rangle_B \cdot y \cdot B^m$. For smaller $i$, we have $x^{(i-1)} = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$.

For all $i$, we then have the invariants ($I_1$) $x^{(i)} + \langle [q_m, \ldots, q_i] \rangle_B \cdot y \cdot B^i = x$ and ($I_2$) $0 \leq x^{(i)} < |y| \cdot B^i$. Therefore, the remainder $r$ is the $B$-complement number $r = x^{(0)}$, while the quotient $q$ can be obtained by $B$-complement conversion (Figure B.17) of the number $q = \langle [q_m, \ldots, q_0] \rangle_B$.

**In variant ($I_2$) is seen as follows:** For the induction base $i = m$, we consider three cases:

$x \geq 0$: In this case, we choose $q_m = 0$ so that $x^{(m)} = x$ holds. Then, ($I_2$) can be derived as follows:

- $x^{(m)} = x \geq 0$ by the assumption of this case
- $x^{(m)} = x \leq \text{MaxInt}(m) = B^{m-1} - 1 < B^{m-1} \leq |y| \cdot B^{m-1} \leq |y| \cdot B^m$

$x < 0 \land y > 0$: In this case, we choose $q_m = 1$ so that $x^{(m)} = x + y \cdot B^m$ holds. Then, ($I_2$) can be derived as follows:

- $x^{(m)} = x + y \cdot B^m \geq \text{MinInt}(m) + 1 \cdot B^m = B^{m-1} - 1 + |y| \cdot B^m < |y| \cdot B^m$

$x < 0 \land y < 0$: In this case, we choose $q_m = -1$ so that $x^{(m)} = x - y \cdot B^m$ holds. Then, ($I_2$) can be derived as follows:

- $x^{(m)} = x - y \cdot B^m \geq \text{MinInt}(m) - (-1) \cdot B^m = B^{m-1} - 1 + |y| \cdot B^m < |y| \cdot B^m$
Hence, invariant $I_2$ holds for $i = m$. It remains to prove the induction step for invariant $I_2$, which is done similar to the radix-$B$ division: By definition, we have $x^{(i-1)} = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$, i.e., $(F1) \ x^{(i)} = x^{(i-1)} + q_{i-1} \cdot y \cdot B^{i-1}$.

Now consider two cases to prove $x^{(i-1)} < |y| \cdot B^{i-1}$ (the second part of $I_2$):

$y > 0$: In this case $q_{i-1} \geq 0$, and by the maximal choice of $q_{i-1}$, we conclude $(F2) \ (q_{i-1} + 1) \cdot y \cdot B^{i-1} > x^{(i)}$. By $(F1)$ and $(F2)$, it follows that $x^{(i-1)} + q_{i-1} \cdot y \cdot B^{i-1} < (q_{i-1} + 1) \cdot y \cdot B^{i-1}$ holds, which is equivalent to $x^{(i-1)} < y \cdot B^{i-1} = |y| \cdot B^{i-1}$.

$y < 0$: In this case $q_{i-1} \leq 0$, and by the minimal choice of $q_{i-1}$, we conclude $(F3) \ (q_{i-1} - 1) \cdot y \cdot B^{i-1} > x^{(i)}$. By $(F1)$ and $(F3)$, it follows that $x^{(i-1)} + q_{i-1} \cdot y \cdot B^{i-1} < (q_{i-1} - 1) \cdot y \cdot B^{i-1}$ holds, which is equivalent to $x^{(i-1)} < -y \cdot B^{i-1} = |y| \cdot B^{i-1}$.

It remains to prove that $0 \leq x^{(i-1)}$ holds, which is however a simple consequence of condition (3) of the choice of $q_{i-1}$: Since (3) $q_{i-1} \cdot y \cdot B^{i-1} \leq x^{(i)}$ holds, we have $0 \leq x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1} \ (F1) = x^{(i)}$.

Due to $(I_1)$ and $(I_2)$, we have in particular $x^{(0)} + [\langle q_m, \ldots, q_0 \rangle]_B \cdot y = x$ and $0 \leq x^{(0)} < |y|$, so that by Theorem B.22, we conclude that $(x \div y) = \langle [q_m, \ldots, q_0] \rangle_B$ and $(x \mod y) = x^{(0)}$.

\[ \square \]

As can be seen, the algorithm described in the above theorem computes a ‘quotient’ whose digits are potentially negative digits while the remainder is already in the desired form. Moreover, from condition (2), it follows that for $y > 0$, the quotient is already a valid $B$-complement number, while for $y < 0$, the quotient has digits $q_i \in \{-(B-1), \ldots, 0\}$. It is not difficult to see that in case $y < 0$, the algorithm performs essentially a division by $-y$ and negates the quotient digits, since by the previous theorem, we have already seen that $(x \div (-y)) = -(x \div y)$ and $(x \mod (-y)) = (x \mod y)$ hold. While the equation $-\langle [q_m, \ldots, q_0] \rangle_B = \langle [-q_m, \ldots, -q_0] \rangle_B$ is valid, the latter does not generate a $B$-complement number due to the potentially negative digits.

The final conversion of the quotient to a $B$-complement number is however not difficult: All we need to do is to apply the algorithm to convert to $B$-complement numbers as shown in Figure B.17. Note that we already know that $m + 1$ digits are sufficient for the quotient. For example, the division of the 10-complement number 45678 by 33 (which is division of -34322 by -27 in usual decimal encoding) is done as follows:
19965678 = x^{(5)}
33 * -1 \implies q[5] = -1
---
026
02665678 = x^{(4)}
33 * -9 \implies q[4] = -9
---
023
00235678 = x^{(3)}
33 * -8 \implies q[3] = -8
---
019
00019678 = x^{(2)}
33 * -7 \implies q[2] = -7
---
007
00000778 = x^{(1)}
33 * -2 \implies q[1] = -2
---
023
00000238 = x^{(0)}
33 * -8 \implies q[0] = -8
---
022

Note that we first have to make a digit extension to perform the shifted addition of the first partial product, so that the numbers $x^{(i)}$ were calculated with 8 digits each. Recall further that the first partial product is added, while the remaining ones are subtracted. The preliminary quotient $[-1, -9, -8, -7, -2, -8]$ is then converted to the correct 10-complement number $\langle 0, 1, 2, 7, 2 \rangle_{10} = 1272$, while the remainder $\langle 0, 2, 2 \rangle_{10} = 22$ is already given in 10-complement.

The previous two theorems are the basis of the $B$-complement division algorithm shown in Figure B.21. The algorithm is similar to the corresponding one for radix-$B$ numbers given in Figure B.8, in particular, we store the values $x^{(i)}$ also in the variable $z$ and perform the subtractions of the partial products from that variable. Several minor points should be emphasized on that algorithm:

- Function $\text{DgtSubBComp1}$ must take care of the most significant digits of $y$ and $z$, since these digits count negative (as for all $B$-complement numbers). While $y[n-1]$ is used in every call to $\text{DgtSubBComp1}$, $z[n+m]$ is only reached in the first call of $\text{DgtSubBComp1}$ (since the remaining calls refer to a subsequence $z[n+i..1]$ where $i < m$ holds).
- The first call to $\text{DgtSubBComp1}$ must perform an addition instead of a subtraction to compute $x^{(m)} := x + q_m \cdot y \cdot B^m$. We determine the most signifi-
function DgtSubBCompl(bool negz, digit z[n+1], q, y[n]) {
    // compute \( s[n..0] = z[n..0] - q \cdot y[n-1..0] \)
    digit cs, sm, s[n+1];
    cs = 0;
    for(int j=0..n-2) {
        sm = cs + z[j] - q * y[j];
        s[j] = sm mod B;
        cs = sm div B;
    }
    // note that \( y[n-1] \) counts negatively
    sm = cs + z[n-1] + q * y[n-1];
    s[n-1] = sm mod B;
    cs = sm div B;
    // note that \( z[n] \) may count negatively
    s[n] = (negz?(cs - z[n]):(cs + z[n]));
    return s;
}

function IntDivMod0(digit x[m], y[n]) {
    digit cs, sm, z[m+n+1], s[n+1], q[m+1], r[n+1], sign;
    z = DigitExtend(x, m+n);
    // negative quotient digits for negative divisor \( y \)
    sign = (y[n-1]==0)?(+1):(-1);
    q[m] = (x[m-1]==0)?0:sign;
    s = DgtSubBCompl(true, z[n+m..m], -q[m], y);
    z[n+m..m] = s[n..0];
    for(int i=m-1..0) {
        // try digits \( B-1,B-2,...,0 \) for \( q[i] \)
        q[i] = B;
        do {
            q[i] = q[i] - 1;
            s = DgtSubBCompl(false, z[n+i..i], (sign*q[i]), y);
            // now \( s[n..0] = z[n+i..i] - (sign*q[i]) \cdot y[n-1..0] \)
        } while(s[n]!=0); // repeat if \( z[n+i..i] < q[i] \cdot y[n-1..0] \)
        z[n+i..i] = s[n..0];
        q[i] = sign*q[i];
    }
    r[n..0] = z[n..0];
    // adjust potentially negative quotient digits
    for(int i=0..m) {
        sm = (i==m)?(cs - q[i]):(cs + q[i]);
        q[i] = sm mod B;
        cs = sm div B;
    }
    q[m] = (cs==-1)?(B-q[m]):q[m];
    return (q, r);
}

Fig. B.21. Division of \( B \)-Complement Numbers
The remaining calls to DgtSubBCompl perform the subtractions $x^{(i-1)} := x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}$. We try the digits $B-1, B-2, \ldots, 0$ until a positive sum is obtained (as required by constraint (3) of the previous theorem). However, as $q_{i-1} \leq 0$ should hold for $y < 0$, we use $\text{sign} \cdot q_{i}$ for the computations, and also finally assign this value to $q_{i}$ once the correct digit has been found.

Finally, the quotient has to be adjusted in case there are negative digits. To this end, we essentially apply function $B\text{Complement}$ shown in Figure B.17 on page 394. However, there is again one special issue: as we know that the quotient fits into $m$ digits, there is no need to add a final digit at position $m+1$ (as would be done by a direct call to function $B\text{Complement}$). Instead, we directly perform a reduction to $m$ digits in case a final carry -1 is left (note that the final carry must be either 0 or -1).

Having a first implementation of the $B$-complement division algorithm, we can now go on with potential improvements, and to this end, we could follow the same proposals we considered for the improvements of the radix-$B$ division algorithm. The first simple improvement is to eliminate the local variable $z$, since we only need to store $n + 1$ digits $z[n+i..i]$ since the digits in $z[n+m..n+i+1]$ are all zero and the digits in $z[i-1..0]$ are those from $x[i-1..0]$. Hence, we can also store the $n + 1$ digits $z[n+i..i]$ in the remainder $r$ as shown in Figure B.22.

Finally, it is now straightforward to extend this algorithm to a restoring and a non-restoring version that does not have the need for a subtraction/addition combined with a multiplication with a digit as performed by DgtSubBCompl.

We defined $B$-complement division such that for given integers $x$ and $y$, we have to compute integers $q$ and $r$ such that the conditions (1) $x = q \cdot y + r$ and (2) $0 \leq r < |y|$ hold. While condition (1) is not to be questioned, there are alternatives to condition (2) that we discuss in the following. The most general algebraic structure where division is defined are Euclidean rings. An Euclidean ring (Euclidean domain) is an integral domain$^2$ that is endowed

\[ \forall a, b, c \in \mathbb{R}. \ (a \oplus b) \odot c = a \odot (b \oplus c) \]
\[ \forall a, b \in \mathbb{R}. \ a \oplus b = b \oplus a \]
\[ \forall a \in \mathbb{R}. \ 0 \oplus a = a \]
\[ \forall a \in \mathbb{R}. \exists \overline{a} \in \mathbb{R}. \ a \oplus \overline{a} = 0 \]
\[ \forall a, b, c \in \mathbb{R}. \ (a \odot b) \odot c = a \odot (b \odot c) \]
\[ \forall a, b \in \mathbb{R}. \ a \odot b = b \odot a \]
\[ \forall a \in \mathbb{R}. \ 1 \odot a = a \]
\[ \forall a, b \in \mathbb{R}. \ a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \]
\[ \forall a, b, c \in \mathbb{R}. \ (a \odot b) \odot c = (a \odot c) \oplus (b \odot c) \]
function IntDivMod1(digit x[m], y[n]) {
    digit cs, sm, s[n+1], q[m+1], r[n+1], sign;
    r = DigitExtend(x, m+n)[n+m..m];
    x = DigitExtend(x, m+n)[m-1..0];
    // negative quotient digits for negative divisor y
    sign = (y[n-1] == 0) ? (+1) : (-1);
    q[m] = (x[m-1] == 0) ? 0 : sign;
    r = DgtSubBCompl(true, r, -q[m], y);
    for(int i=m-1..0) {
        // x(i+1) is stored in r[n-1..0]@x[i..0]
        // compute x(i) in r[n-1..0]@x[i-1..0]
        // to this end, try digits B-1, B-2, ..., 0 for q[i]
        q[i] = B;
        do {
            q[i] = q[i] - 1;
            s = DgtSubBCompl(false, r[n-1..0]@x[i], (sign*q[i]), y);
        } while(s[n] != 0);
        r = s;
        q[i] = sign*q[i];
    }
    // adjust potentially negative quotient digits
    for(int i=0..m) {
        sm = (i==m) ? (cs - q[i]) : (cs + q[i]);
        q[i] = sm mod B;
        cs = sm div B;
    }
    q[m] = (cs==-1) ? (B-q[m]) : q[m];
    return (q, r);
}

Fig. B.22. Improved Version of Division of B-Complement Numbers

by a norm, i.e. a function that maps elements x of the ring to non-negative integers |x|, so that for all elements x, y with y \neq 0, there are elements q, r such that (1) x = q \cdot y + r and (2) either r = 0 or |r| < |y|.

However, conditions (1) and (2) do not uniquely determine the quotient q and the remainder r as shown in the following lemma. The lemma furthermore tells us that there are not too many solutions:

An integral domain is a commutative ring with the zero-product property:

- \forall a, b \in \mathbb{R}, a \neq 0 \land b \neq 0 \rightarrow a \odot b \neq 0

Hence, the product of any two non-zero elements is always non-zero; that is there are no zero divisors.
Lemma B.33. Given integers $x$ and $y \neq 0$, consider the set $D_{x,y}$ of pairs of integers $(q,r)$ that satisfy (1) $x = q \cdot y + r$ and (2) $|r| < |y|$. Then, one of the following two cases holds for integers $(x,y)$:

- $D_{x,y} = \{(q,0)\}$, i.e., quotient and remainder are uniquely determined.
- $D_{x,y} = \{(q_1,r_1),(q_2,r_2)\}$, where $q_2 = q_1 + 1$ and $r_2 = r_1 - y$. Moreover, one of the possible remainders is positive, while the other one is negative.

Proof. By condition (1) and (2), it follows $|r| = |x - q \cdot y| < |y|$, which is equivalent to (3) $q \cdot y \in \{x - (|y| - 1), \ldots, x + (|y| - 1)\}$. Hence, $q \cdot y$ is inside a $|y| - 1$ sphere’ around $x$.

Assume that there are two solutions $(q_1,r_1)$ and $(q_2,r_2)$ satisfying conditions (1) and (2). By condition (1), it follows $q_1 \cdot y + r_1 = q_2 \cdot y + r_2$, thus $r_2 - r_1 = (q_1 - q_2) \cdot y$, and by condition (2), we further conclude $|r_1 - r_2| = |(q_1 - q_2) \cdot y| < 2 |y|$, so that (4) $|q_1 - q_2| < 2$ follows for all solutions $(q_1,r_1)$ and $(q_2,r_2)$.

Without loss of generality, assume therefore that $q_2 = q_1 + 1$ holds. It follows then that $r_2 = r_1 - y$ holds. However, if $r_1$ should be zero, then $r_2$ would violate condition (2). Hence, in case (1) and (2) should be satisfied with a pair $(q,0)$, then this is the only solution of (1) and (2).

Now, assume $r \neq 0$, and assume that there is a third solution $(q_3,r_3)$. Due to (4), we have $|q_1 - q_3| < 2$, and since $q_3$ should be different to $q_1$ and different to $q_2$, it follows $q_3 = q_1 - 1$, and thus $r_3 = r_1 + y$. However, then $|r_3 - r_2| = 2 |y|$, which is not possible.

In our previous definition of the quotient and the remainder, we have chosen the uniquely determined pair $(q,r)$ where $r$ is non-negative, that is, we added the third condition (3) $0 \leq r$. Alternatively, one could demand as a third condition (3)$'$ $\text{sign}(x) = \text{sign}(r)$ (or $r = 0$). Using this condition also uniquely determines a quotient and a remainder, that are used in many programming languages and microprocessors. This alternative definition has the following nice properties: $(x \text{ div } (\neg y)) = ((\neg x) \text{ div } y) = (\neg (x \text{ div } y))$, $(\neg (x \text{ div } y)) = (x \text{ div } y)$, $(x \text{ mod } (\neg y)) = (x \text{ mod } y)$, $(\neg (x \text{ mod } y)) = ((\neg x) \text{ mod } (\neg y)) = (\neg (x \text{ mod } y))$.

While these sign rules are attractive, they do not simplify the division algorithms. Even the reduction to radix-$B$ division does not become simpler, since changing the sign is similar to incrementing the number. Also the direct $B$-complement division algorithm is very similar: As the numbers $x^{(i)}$ are the preliminary remainders, they have to maintain the initial sign of $x$. For this reason, the strategy of this algorithm is to drive $x^{(i)}$ towards zero, but maintaining the initial sign (instead of first making $x^{(m)}$ positive).

Before presenting the construction, let us determine the maximal number of required digits for the alternative definitions. For the quotient, we have again the worst case example $(\text{Minint}(m) \text{ div } (-1)) = -\text{Minint}(m)$ that still requires $m + 1$ digits. As the remainder may now become negative, the worst
case is now \(-|y|\), which requires one more digit than \(y\). Hence, the number of required digits remains the same for our alternative definition.

**Theorem B.34 (Alternative Division of B-Complement Numbers).**

For \(B\)-complement numbers \(x = \langle \langle x_{m-1}, \ldots, x_0 \rangle_B^Z \rangle_B^Z\) and \(y = \langle \langle y_{n-1}, \ldots, y_0 \rangle_B^Z \rangle_B^Z\), we define the \(m+1\) \(B\)-complement numbers \(x^{(i)}\) and the \(m+1\) digits \(q_i\) recursively as follows:

- \(x^{(m)} \overset{\text{def}}{=} x + q_m \cdot B^m\), where \(q_m := 0\).
- \(x^{(i-1)} := x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}\) for \(i = m, \ldots, 1\), where \(q_{i-1}\) is chosen from \((-B-1), \ldots, 0, \ldots, B-1\) such that (1) \(|q_{i-1}|\) is maximal, (2) \(|x^{(i-1)}|\) is minimal, and that (3) \(\text{sign}(x^{(i-1)}) = \text{sign}(x^{(i)})\) holds.

By this construction, the following invariants hold for all \(i\):

- \((I_1)\) \(x^{(i)} + \langle \langle q_m, \ldots, q_i \rangle_B^Z \rangle_B^Z \cdot y \cdot B^i = x\)
- \((I_2)\) \(|x^{(i)}| < |y| \cdot B^i\)
- \((I_3)\) \(\text{sign}(x^{(i)}) = \text{sign}(x)\)

Therefore, for the quotient \(q\) that can be obtained by \(B\)-complement conversion (Figure B.17) and the remainder \(r := x^{(0)}\), we have:

1. \(x = \langle \langle q_m, \ldots, q_0 \rangle_B^Z \rangle_B^Z \cdot y + r\)
2. \(|r| < |y|\)
3. \(\text{sign}(x) = \text{sign}(r)\)

**Proof.** We first prove invariant \((I_1)\): For \(i = m\), we have \(x^{(m)} = x + q_m \cdot y \cdot B^m = x - \langle \langle q_m \rangle_B^Z \rangle_B^Z \cdot y \cdot B^m\). For smaller \(i\), we have \(x^{(i-1)} = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1}\) \(\overset{\text{def}}{=} \langle \langle q_{m-1}, \ldots, q_{i-1} \rangle_B^Z \rangle_B^Z \cdot y \cdot B^{i-1}\) so that invariant \((I_1)\) holds.

Invariant \((I_2)\) is seen as follows: For the induction base \(i = m\), the proof is trivial: \(|x^{(m)}| = |x| \leq (B - 1) \cdot B^{m-1} < B^m \leq |y| \cdot B^m\). For the induction step, we consider four cases:

- \(x \geq 0, y > 0\): In this case \(q_{i-1} \geq 0\) holds (since \(q_{i-1} < 0\) makes \(q_{i-1} \cdot y \) negative, so that \(|x^{(i-1)}| \geq |x^{(i)}|\) would follow). Due to the maximal choice of \(|q_{i-1}|\), it follows that \(q_{i-1} + 1\) would violate condition (3), i.e., we have \(x^{(i)} - (q_{i-1} + 1) \cdot y \cdot B^{i-1} < 0\) which is equivalent to \(|x^{(i-1)}| = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1} < y \cdot B^{i-1} = |y| \cdot B^{i-1}\).
- \(x \geq 0, y < 0\): In this case \(q_{i-1} \leq 0\) holds (since \(q_{i-1} > 0\) makes \(q_{i-1} \cdot y \) negative, so that \(|x^{(i-1)}| \geq |x^{(i)}|\) would follow). Due to the maximal choice of \(|q_{i-1}|\), it follows that \(q_{i-1} - 1\) would violate condition (3), i.e., we have \(x^{(i)} - (q_{i-1} - 1) \cdot y \cdot B^{i-1} < 0\) which is equivalent to \(|x^{(i-1)}| = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1} < -y \cdot B^{i-1} = |y| \cdot B^{i-1}\).
- \(x \leq 0, y > 0\): In this case \(q_{i-1} \leq 0\) holds (since \(q_{i-1} > 0\) makes \(q_{i-1} \cdot y \) positive, so that \(|x^{(i-1)}| \geq |x^{(i)}|\) would follow). Due to the maximal choice of \(|q_{i-1}|\), it follows that \(q_{i-1} - 1\) would violate condition (3), i.e., we have \(x^{(i)} - (q_{i-1} - 1) \cdot y \cdot B^{i-1} > 0\) which is equivalent to \(|x^{(i-1)}| = x^{(i)} - q_{i-1} \cdot y \cdot B^{i-1} > 0\).
\[ -y \cdot B^{i-1}, \text{ which in turn is equivalent to } |x^{(i-1)}| = -(x(i) - q_{i-1} \cdot y) = -y \cdot B^{i-1} = |y| \cdot B^{i-1} \]

\( x \leq 0, y < 0 \): In this case \( q_{i-1} \geq 0 \) holds (since \( q_{i-1} < 0 \) makes \( q_{i-1} \cdot y \) positive, so that \( |x^{(i-1)}| \geq |x(i)| \) would follow). Due to the maximal choice of \( |q_{i-1}| \), it follows that \( q_{i-1} + 1 \) would violate condition (3), i.e., we have \( x^{(i)} - (q_{i-1} + 1) \cdot y \cdot B^{i-1} \geq 0 \) which is equivalent to \( y \cdot B^{i-1} \), i.e., \( -x^{(i-1)} < -y \cdot B^{i-1} \), and thus \( |x^{(i-1)}| = -x^{(i-1)} < -y \cdot B^{i-1} \).

It remains to prove invariant \( I_3 \), which is trivial for \( i = m \) and very simple in the induction base, since that can be directly concluded from the choice of \( q_{i-1} \).

\( \square \)

Hence, only slight modifications in terms of the digit selection procedure have to be made to adapt the previous algorithm in order to compute the alternative quotient and remainder. Note that also in the alternative algorithm, the quotient digits are either all non-positive or all non-negative, depending on the signs of the dividend and the divisor (quotient digits are non-negative iff signs of dividend and divisor are the same).

\section*{B.5 Hardware-Implementations}

In the previous sections, we have presented the theory of radix-\( B \) and \( B \)-complement numbers together with the related algorithms to perform the basic arithmetic operations. This is required to define the semantics of the Quartz language, and also for software synthesis from Quartz programs to be able that arithmetic with arbitrarily many digits can be performed independently of the underlying microprocessor’s bitwidth.

For the hardware synthesis of Quartz programs, these algorithms have to be considered for the case \( B = 2 \), so that the digits are \( \{0, 1\} \) which can be represented in hardware circuits with boolean values. The digit arithmetic that was used in the algorithms so far can then be replaced with propositional logic expressions. In this section, we briefly consider these ‘propositional versions’ of these algorithms that are obtained in the special case of \( B = 2 \).

\subsection*{B.5.1 Radix-2 Arithmetic}

The algorithm for comparing two given radix-2 number is very simple: equality testing is already very simple in the general case. For checking \( \langle x_{n-1}, \ldots, x_0 \rangle_2 \prec \langle y_{n-1}, \ldots, y_0 \rangle_2 \), we compute the propositional formula \( \ell_{n-1} \) with the following recursion:

\begin{itemize}
  \item \( \ell_0 := \lnot x_0 \land y_0 \)
  \item \( \ell_{i+1} := \lnot x_{i+1} \land y_{i+1} \lor (x_{i+1} \leftrightarrow y_{i+1}) \land \ell_i \)
\end{itemize}
module BinLess(bool x[n], y[n], &ls) {
    bool l[n];
    l[0] = !x[0] & y[0];
    for (nat<n> i=1..n-1)
        l[i] = !x[i] & y[i] | (x[i]->y[i]) & l[i-1];
    ls = l[n-1];
}

module BinEqual(bool x[n], y[n], &eq) {
    bool e[n];
    e[0] = x[0]<->y[0];
    for (nat<n> i=1..n-1)
        e[i] = (x[i]<->y[i]) & e[i-1];
    eq = e[n-1];
}

Fig. B.23. Comparison of Radix-2 Numbers

The Quartz modules given in Figure B.23 implement the comparison of binary numbers such that the comparisons are done in one cycle. Hence, a combinational circuit of depth $O(n)$ will by obtained by the hardware synthesis. Clearly, by adding pause-statements, these algorithms can be made sequential, and by reorganizing the for-loops, one can generate a tree for with depth $O(\log(n))$ instead of the combinational circuit of depth $O(n)$.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>c</th>
<th>$x+y+c$</th>
<th>$x \land y \lor c \land (x \lor y)$</th>
<th>$x \oplus y \oplus c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$[0,0]_B$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$[0,1]_B$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$[0,1]_B$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$[1,0]_B$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$[0,1]_B$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$[1,0]_B$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$[1,0]_B$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[1,1]_B$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.3. Correctness of Binary Addition

To implement addition and subtraction, we have to encode the addition and subtraction algorithms of Figure B.4 in propositional logic. For the addition algorithm, we have to find propositional formulas for the expressions $(x[i]+y[i]+c) \text{ div } 2$ and $(x[i]+y[i]+c) \text{ mod } 2$. Table B.3 shows possible propositional formulas for these expressions, so that our propositional algorithm for adding binary numbers given in Figure B.24 is obtained.
module BinNatAdd(bool x[n], y[n], &s[n], &cout) {
    event c[n+1];
    c[0]=false;
    for(int i=0..n-1) {
        s[i] = x[i] xor y[i] xor c[i];
        c[i] = x[i] & y[i] | c[i] & (x[i] | y[i]);
    }
    cout = c[n];
}

function BinNatSub(bool x[n], y[n], &s[n], &cout) {
    event c[n+1];
    c[0]=true;
    for(int i=0..n-1) {
        s[i] = x[i] xor !y[i] xor c[i];
        c[i] = x[i] & !y[i] | c[i] & (x[i] | !y[i]);
    }
    cout = c[n];
}

Fig. B.24. Addition and Subtraction of Radix-\(B\) Numbers

For the implementation of the subtraction algorithm, we make use of the addition of the complement to avoid to already deal with negative numbers. Table B.4 proves that we can make of the expressions \(x_i \land \neg y_i \lor c_i \land (x_i \lor \neg y_i)\) and \(x_i \oplus \neg y_i \oplus c_i\) to implement the algorithm BinNatSub as shown in Figure B.24.

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
<th>(v)</th>
<th>(v\lor \neg y\lor c)</th>
<th>(v\land \neg y\land c)</th>
<th>(v\lor (x\lor \neg y))</th>
<th>(v\oplus \neg y\oplus c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.4. Correctness of Binary Subtraction

For the implementation of both the binary addition and subtraction algorithm, we use essentially the same basic circuit that is called a full adder. The following lemma explains the implementation of such a full-adder circuit:
Lemma B.35 (Full Adders). Given bits $x$, $y$, and $c_{in}$, we have for $s := x \oplus y \oplus c_{in}$ and $c_{out} := x \land y \lor c_{in} \land (x \lor y)$ the following equation:

$$\langle [x] \rangle_2^N + \langle [y] \rangle_2^N + \langle [c_{in}] \rangle_2^N = \langle [s, c_{out}] \rangle_2^N$$

Thus, a full-adder circuit with inputs $x$, $y$, and $c_{in}$ and outputs $s$ and $c_{out}$ as defined above can be used to add three bits to a two-bit radix-2 number.

A simple implementation of an adder can therefore be obtained by constructing a cascade of full adders as shown in Figure ?? which yields a so-called carry-ripple adder. The problem of a carry-ripple adder is that the depth of the circuit, i.e., the length of the longest path of the circuit is $O(n)$, so that the delay time of the circuit grows linearly with the number of bits. Using radix-$B$ representations this can in general not be avoided, and for this reason, there are other number representations like those that we consider in Section B.6 where addition circuits with depth $O(1)$ can be obtained. On the good side, even the carry-ripple adder requires only $O(n)$ gates to add two $n$-bit numbers.

From the practical side, there are alternatives to the carry-ripple adder like the carry-select, carry-save, and carry-lookahead adders. All of these implementations do also have depth $O(n)$, but they improve the depth by some constant.

For the subtraction, we can use the same circuits as for addition, except that we have to negate the bits of the second operand and that we have to start with an initial carry bit that is true.

The next operation we want to implement is the multiplication. Considering the algorithm given in Figure B.7, we see that we need propositional formulas to compute number $s = p[i+j] + x[j] \cdot y[i] + c$. It is clear that
we need two formulas since the result value may require two bits for its radix-2 representation. These two propositional expressions can be derived from Table B.3 by simply adding \( p \), \( x \land y \) and \( c \) as shown in Figure B.26 by module MultCell. Module BinNatMult then simply instantiates such cells in an array to perform the additions of the partial products.

```plaintext
class MultCell(bool p, x, y, c, &p1, &p0) {
    event xy;
    xy = x & y;
    p1 = x & xy | c & (x | xy);
    p0 = x xor xy xor c;
}
class BinNatMult(bool x[m], y[n], &p[m+n]) {
    event s[n][m+1], c[n][m];
    for(int i=0..n-2) {
        for(int j=0..m-1)
            MultCell(s[i][j+1], x[j], y[i], c[i][j], s[i+1][j], c[i][j+1]);
        s[i+1][m] = c[i][m];
        p[i] = s[i+1][0];
    }
    for(int j=0..m-1)
        MultCell(s[n-1][j+1], x[j], y[n-1], c[n-1][j], p[n-1], c[n-1][j+1]);
}
```

**Fig. B.26.** Multiplication of Radix-2 Numbers

Note that the array multiplier given in Figure B.26 is a bit wasteful: there are no assignments to \( s[0][0..0] \), and therefore these values are false. Therefore the first row of multiplication cells could be simplified since their \( p \) arguments are all zero.

Finally, let us consider the division operation. Clearly, it is again the most complex operation from the four basic arithmetic operations. Fortunately, the special case \( B = 2 \) simplifies the algorithm significantly: In particular, there is no pressing need for quotient digit estimation, since we can simply test the two possible digits. The result is then the simple non-performing division algorithm shown in Figure B.27.

The algorithm is called ‘non-performing’ since it stores the result of the subtraction in a separate variable and only after checking its sign forwards this result to the preliminary remainder \( r \) or not. The more popular versions are the restoring and the non-restoring division algorithms that do not require the local variable \( s \).

The restoring division is obtained from the non-performing division by omitting the local variable \( s \) and using the variable \( r \) instead for storing the
function BinDivMod(digit x[m], y[n]) {
    // maintain x(i) in r[n-1..0]@x[m-1..0]
    digit q[m], qd, s[n+1], r[n], c[n];
    for(int i=m-1..0) {
        // try quotient bit qd=true
        qd = true;
        // compute s[n..0]@x[i-2..0] = x(i)-y*B**(i-1)
        // note that x(i) is r[n-1..0]@x[i-1..0]
        s[0] = x[i-1] xor y[0];
        c[0] = x[i-1] or y[0];
        for(int j=0..n-2) {
            s[j+1] = c[j] xor r[j] xor (not y[j+1]);
            c[j+1] = r[j] and not y[j+1] | c[j] & (r[j] or not y[j+1]);
        }
        s[n] = c[n-1] xor r[n-1];
        // s is positive iff s[n]=false
        q[i] = not s[n];
        if(q[i])
            for(int j=0..n-1)
                r[j] = s[j];
        return (q, r);
    }
}

Fig. B.27. Non-Performing Radix-2 Division

result of the subtraction. In iteration $i$, we compute $z = x^{(i)} - y \cdot 2^{i-1}$ to check whether this is the correct value of $x^{(i-1)}$. If $z$ is positive, then we have $x^{(i-1)} = z$, and everything is okay. However, if $z$ is negative, then we have $x^{(i-1)} = x^{(i)}$ and we therefore have to restore $x^{(i)}$ from $z$ by computing $x^{(i-1)} = x^{(i)} = z + y \cdot 2^{i-1}$.

The non-restoring division defers the restoration step to the next iteration in that the next iteration performs an addition instead of a subtraction. This is justified as follows: In iteration $i$, we have computed $z = x^{(i)} - y \cdot 2^{i-1}$ to check whether this is the correct value of $x^{(i-1)}$. If $z$ is positive, then we have $x^{(i-1)} = z$, and everything is okay. However, if $z$ is negative, then we have $x^{(i-1)} = x^{(i)}$ and we therefore should restore $x^{(i)}$ from $z$ by computing $x^{(i-1)} = x^{(i)} = z + y \cdot 2^{i-1}$. In the non-restoring division, we combine the restoration with the next subtraction as follows: $x^{(i-1)} - y \cdot 2^{i-2} = z + y \cdot 2^{i-1} - y \cdot 2^{i-2} = z + y \cdot 2^{i-2}$, i.e., we simply add in the next iteration the value that would have otherwise been subtracted.

For hardware implementations, the nonrestoring division is preferable. It is better than the non-restoring division since it does not require the restoration step. Whether it is better than the non-performing division depends on the particular translation to hardware gates. If a purely combinational circuit is generated, then the local variables like $s$ do only generate wires, but no
B.5 Hardware-Implementations

 registers. Then, the presence of the local variable \( s \) does not add additional costs. However, in a sequential circuit, we would have to use additional registers which would make the non-performing division more expensive than the nonrestoring division.

B.5.2 2-Complement Arithmetic

Addition and Subtraction

Addition for an argument pair \( \text{nat}[n] \times \text{nat}[m] \) or \( \text{int}[n] \times \text{int}[m] \) first performs zero/sign extension to the argument with fewer bits, so that both argument have the same bitwidth. After this, the well-known binary addition is performed as shown in the following table:

\[
\begin{array}{|c|c|}
\hline
+ : \text{nat}[n] \times \text{nat}[m] \rightarrow \text{nat}[n+1] & + : \text{int}[n] \times \text{int}[m] \rightarrow \text{int}[n+1] \\
\hline
\begin{align*}
 c_0 & := \text{false} \\
 c_{i+1} & := a_i \land b_i \lor c_i \land (a_i \lor b_i) \\
 s_i & := \begin{cases} 
 a_i \oplus b_i \oplus c_i & : \text{for } i < n \\
 c_n & : \text{for } i = n 
\end{cases}
\end{align*} & \begin{align*}
 c_0 & := \text{false} \\
 c_{i+1} & := a_i \land b_i \lor c_i \land (a_i \lor b_i) \\
 s_i & := \begin{cases} 
 a_i \oplus b_i \oplus c_i & : \text{for } i < n \\
 c_n & : \text{for } i = n 
\end{cases}
\end{align*}
\hline
\end{array}
\]

Note that addition on \( \text{nat}[n] \times \text{nat}[m] \) is different to addition on \( \text{int}[n] \times \text{int}[m] \), since the leading sum bit is computed differently (since for generating the \( n + 1 \) sum (which can also be explained by zero/sign extension of the arguments by one further bit). Hence, in both cases, the result requires \( \max((n, m)) + 1 \) bits.

Subtraction is performed similarly, where the same hardware circuit is used, but the initial carry bit is \( \text{true} \) and the bits of the second argument are negated:

\[
\begin{array}{|c|c|}
\hline
- : \text{nat}[n] \times \text{nat}[m] \rightarrow \text{nat}[n+1] & - : \text{int}[n] \times \text{int}[m] \rightarrow \text{int}[n+1] \\
\hline
\begin{align*}
 c_0 & := \text{true} \\
 c_{i+1} & := a_i \land \neg b_i \lor c_i \land (a_i \lor \neg b_i) \\
 s_i & := \begin{cases} 
 a_i \oplus \neg b_i \oplus c_i & : \text{for } i < n \\
 \neg c_n & : \text{for } i = n 
\end{cases}
\end{align*} & \begin{align*}
 c_0 & := \text{true} \\
 c_{i+1} & := a_i \land \neg b_i \lor c_i \land (a_i \lor \neg b_i) \\
 s_i & := \begin{cases} 
 a_i \oplus \neg b_i \oplus c_i & : \text{for } i < n \\
 \neg c_n & : \text{for } i = n 
\end{cases}
\end{align*}
\hline
\end{array}
\]

Clearly, before this operation takes place, we also use zero/sign extension so that both operands will have the same bitwidth. Hence, also subtraction yields a result with \( \max((n, m)) + 1 \) bits. As an example, 4u-1u evaluates to bitvectors whose representations are the same as 100b and 1b, and the result has first the representation of 0011b. However, static expressions are always reduced to the minimal bitwidth, so that the final result has only two bits.

As already explained, addition and substraction for mixed types \( \text{nat}[n] \times \text{int}[m] \) or \( \text{int}[m] \times \text{nat}[n] \) first performs an automatic type conversion for converting the argument with type \( \text{nat}[n] \) to \( \text{int}[n+1] \), and then invokes the addition/subtraction for the resulting argument pair \( \text{int}[n+1] \times \text{int}[m] \). At the end, no bits have to be eliminated from the final result.
Unary Minus

The unary minus operation - is a special case of a subtraction operation in that \(-n\) is replaced with \(0 - n\). Hence, this is defined as follows:

\[
\begin{array}{c|c}
\text{Unary Minus} & \\
\hline 
\text{nat}[n] & \text{int}[n+1] \\
\hline 
c_0 & := \text{true} \\
c_{i+1} & := c_i \land \neg b_i \\
s_i & := \begin{cases} 
-\overline{b_i} \oplus c_i : & \text{for } i < n \\
-\neg c_n & : & \text{for } i = n 
\end{cases}
\end{array}
\]

In general, we need an additional bit to perform this operation. Otherwise, we would have \(-4 == -4\) instead of \(-4 == 4\) since the following hold:

- \(4u == (\text{nat}) 100b\)
- \(-4 == (\text{int}) 100b\)
- \(100b == (\text{bv}) \text{(nat)} \text{not}(100b) + 1\)

The operation on type \(\text{nat}[n] \rightarrow \text{int}[n+1]\) can also be obtained as the addition/subtraction operations above: we first perform a type conversion to replace \((\overline{b_{n-1}}, \ldots, b_0)\) with \((\text{false}, b_{n-1}, \ldots, b_0)\), and apply then the operation on type \(\text{int}[n+1] \rightarrow \text{int}[n+2]\). Clearly, we can then eliminate the leading bit to obtain the correct result of type \(\text{int}[n+1]\).

Multiplication

Multiplication is reduced to addition of partial products with appropriate shifts according to the following equation:

\[
\left( \sum_{i=0}^{m-1} a_i 2^i \right) \left( \sum_{j=0}^{n-1} b_j 2^j \right) = \sum_{j=0}^{n-1} b_j 2^j \left( \sum_{i=0}^{m-1} a_i 2^i \right)
\]

The only difference between signed and unsigned numbers is that for signed numbers the leading bit \(b_{n-1}\) is to be counted negatively, thus performing a subtraction instead of an addition operation. The modules given in Figure B.28 lists a formal definition of these operations (where it is assumed that the operands \(a\) and \(b\) do not change during the sequential multiplication. Of course, the multiplication operation does not take time, so that the algorithms shown in Figure B.28 are just used for illustrating how multiplication is performed.

Note that the addition/subtraction of the partial products has to be made with the full sum, thus yielding one additional bit. This leads at the end to a product whose bitwidth is the sum of the bitwidths of the operands. Note that in contrast to addition/subtraction, there is no need to equalize the bitwidths by zero/sign extensions.

In case argument pairs \(\text{int}[n] \times \text{nat}[m]\) and \(\text{int}[n] \times \text{nat}[m]\) occur, again the argument with type \(\text{nat}[m]\) is converted to type \(\text{int}[m+1]\) before the
Fig. B.28. Definition of the Multiplication Operations

Signed multiplication is used to produce a int[n+m+1] result. However, according to Table ??, the leftmost bit is not necessary, so that we can rely on the lower int[n+m] bits.

Division and Remainder

Similar to multiplication, there are different algorithms to compute x / y and x % y for argument pairs nat[n] × nat[m] and int[n] × int[m]. For arguments of type nat[n] × nat[m], the length of x / y is at most the length of x (due to the expression x/1). For arguments of type int[n] × int[m], the length of x / y is at most the length of x plus one (a worst case example is -4/-1 == (int) 100b / (int) 1b == (int) 0100b == 4).

For mixed types nat[n] × int[m] or int[m] × nat[n], clearly an automatic type conversion from type nat[n] to type int[n+1] is performed before the division with arguments int[n+1] × int[m] and int[m] × int[n+1] is performed. Thus, quotients of type int[n+2] and int[n+1] are obtained. In the former case, however, the leftmost bit is redundant, so that the final result will only have int[n+1] bits.

B.6 Further Representations of Numbers

Besides the radix-\(B\) and the \(B\)-complement numbers, which are both positional number systems, there are also some other representations of numbers
module NatDivision (bv[M] a, &q, bv[N] b, &r) {
    bv[M] q = a;
    bv[N+1] alu;
    bool q_i = true;
    r = 0;
    for(int i=1..N-1) {
        alu = (q_i ? r@q[-1]:b) - b;
        next(q_i) = !(alu[-1]);
        next(r) = alu[-1:];
        next(q) = q[-1:]@!(alu[-1]);
        pause;
    }
    if(!q_i) {
        next(r) = r+rb;
        pause;
    }
}
module IntDivision (bv[M] a, bv[N] b, bv[M+1] &q, bv[N] &r) {
    bv[M] q = a;
    bv[N+1] alu;
    bool q_i = a[-1] <-> b[-1];
    r = a[-1]?-1:0;
    for(int i=1..N-1) {
        alu = (q_i ? r@q[-1]:b) - b;
        next(q_i) = alu[-1] <-> b[-1];
        next(r) = alu[-2:];
        next(q) = q[-1:]@!(alu[-1] <-> b[-1]);
        pause;
    }
    if(b[-0] <-> q_i) {
        if(b[-1])
            next(r) = r-rb;
        else
            next(r) = r+rb;
        pause;
    }
    if(b[-1]) next(q) = q+1;
    pause;
}

Fig. B.29. Definition of the (Nonrestoring) Division Operations

that have their own advantages and disadvantages. Currently, these number representations are not used in Quartz, but may be used in later versions. In particular, residue number systems and signed-digit numbers are prominent candidates to enhance the degree of concurrent computation in the arithmetic
operations. Moreover, Avizienis numbers are often used to increase the degree of fault-tolerance, which is a topic of increasing importance. We therefore already add the main facts about these number systems in this section for future use in the Quartz language.

B.6.1 Gray Code Numbers

B.6.2 Residue Number Systems

B.6.3 Signed Digit Numbers
C

Concrete Grammar of the Quartz Language

C.1 Lexical Tokens

(* ----------- Quartz types ----------------- *)
| "event"  { EVENT } |
| "bool"   { BOOL }  |
| "int"    { INT }   |
| "nat"    { NAT }   |
| "bv"     { BV }    |
| "?"      { QUESTIONMARK } |

(* ----------- Quartz header ----------------- *)
| "module" { MODULE } |
| "implements" { IMPLEMENTS } |
| "end"    { END } |

(* ----------- delimiters and punctuations ----------------- *)
| "{"   { BDEL1 } |
| "}"   { BDEL2 }  |
| "["   { DDEL1 }  |
| "]"   { DDEL2 }   |
| "="   { DDEL2 }  |
| "\"   { DEL1 }   |
| ":"   { DEL2 }   |
| ":\"  { TWOPOINTS } |
| ":\:" { POINT }  |
| ":::"  { TWOCOLON } |
| ":=:"  { COLON }  |
| ":,\" { SEMICOLON } |
| ":,\:" { COMMA }  |

(* ----------- Quartz statements ----------------- *)
| "nothing" { NOTHING } |
| "exit"   { EMIT } |
| "="    { ASSIGN } |
| "next"  { NEXT } |
| "pause" { PAUSE } |
| "if"    { IF } |
| "else"  { ELSE } |
| "||"    { SYNCPAR } |
| "||\"   { ASYNCPAR } |
| "choose" { CHOOSE } |
| "switch" { SWITCH } |
| "do"    { DO } |
| "while" { WHILE } |
| "weak"  { WEAK } |
| "immediate" { IMMED } |
| "suspend" { SUSPEND } |
| "abort" { ABORT } |
| "when"  { WHEN } |
C Concrete Grammar of the Quartz Language

| "try"  { TRY} |
| "throw" { THROW} |
| "let"  { LET} |
| "in"   { IN} |
| "now"  { NOW} |
| "during" { DURING} |
| "holds" { HOLDS} |
| "abstract" { ABSTRACT} |
| "spec" { SPEC} |
| "halt" { HALT} |
| "loop" { LOOP} |
| "each" { EACH} |
| "every" { EVERY} |
| "await" { AWAIT} |
| "suspend" { SUSTAIN} |
| "to" { TO} |
| "sequence" { GENSEQUENCE} |
| "parallel" { GENSYNCPAR} |
| "asyncpar" { GENASYNCPPAR} |
| "interleave" { GENINTERLEAVE} |
| "instance" { INSTANCE} |
| "of" { OF} |
| "base" { BASE} |
| "recursion" { RECURSION} |
| "with" { WITH} |

(* ---- boolean operators ------- *)
| "not"  { NOT} |
| "and"  { AND} |
| "xor"  { XOR} |
| "or"   { OR} |
| "imply" { IMP} |
| "equiv" { EQU} |

(* ---- generic expressions/quantifiers ------- *)
| "forall" { FORALL} |
| "exists" { EXISTS} |
| "sum" { LISTSUM} |

(* ---- numeric relations ------- *)
| "eqq"  { EQQ} |
| "neq"  { NEQ} |
| "leq"  { LEQ} |
| "geq"  { GEQ} |

(* ---- numeric operators ------- *)
| "plus" { PLUS} |
| "minus" { MINUS} |
| "mul"  { MUL} |
| "div"  { DIV} |
| "mod"  { MOD} |

(* ---- bitvector list operators ------- *)
| "reverse" { REVERSE} |
| "append" { APPEND} |

(* ---- static operators ------- *)
| "maxNat" { MAXNAT} |
| "minInt" { MININT} |
| "maxInt" { MAXINT} |
| "exp2" { EXP2} |
| "log2" { LOG2} |
| "lengthOf" { LENGTHOF} |
| "sizeOf" { SIZEOF} |

(* ---- bitvector cast operators ------- *)
C.2 Grammar of the Quartz Language

1 QrzProgram : module_list EOF
Concrete Grammar of the Quartz Language

module_list : Qmodule
  | Qmodule module_list

Qmodule : MODULE ID DEL1 decl_list DEL2 BDEL1 LocStmt BDEL2
  | MODULE ID DEL1 decl_list DEL2
  | IMPLEMENTS impl_list BDEL1 LocStmt BDEL2
  | SPEC ID DEL1 decl_list DEL2 BDEL1 SpecList BDEL2

decl_list : Qtype dfid_list
  | Qtype dfid_list_comma decl_list

Qtype : EVENT
  | BOOL
  | INT
  | INT DDEL1 expr DDEL2
  | NAT DDEL1 expr DDEL2
  | BV DDEL1 expr DDEL2

dfid_list : dfid
  | dfid COMMA dfid_list

dfid_list_comma : dfid COMMA
  | dfid COMMA dfid_list_comma

dfid : id dim_list
  | AND id dim_list
  | QUESTIONMARK id dim_list

dim_list :
  | DDEL1 expr DDEL2 dim_list

id : ID

impl_list : id DEL1 expr_list DEL2
  | id DEL1 expr_list DEL2 COMMA impl_list

SpecList : id COLON expr SEMICOLON
  | id COLON expr SEMICOLON SpecList
  | id DEL1 ID DEL2 COLON expr SEMICOLON
  | id DEL1 ID DEL2 COLON expr SEMICOLON SpecList

Action : EMIT id
  | EMIT NEXT DEL1 id DEL2
  | expr ASSIGN expr
  | NEXT DEL1 expr DEL2 ASSIGN expr
  | NOW DEL1 expr DEL2

case_list :
  | dexpr DO Stmt case_list

expr : DEL1 expr DEL2

dexprs : dexpr SEMICOLON

Generic : DEL1 Qtype id ASSIGN expr TWOPOINTS expr WITH expr DEL2
  | DEL1 Qtype id ASSIGN expr TWOPOINTS expr DEL2

LocStmt : decl_list SEMICOLON LocStmt
  | SeqStmt

SeqStmt : Stmt
  | Stmt SeqStmt

Stmt : NOTHING SEMICOLON
  | Action SEMICOLON
  | id COLON PAUSE SEMICOLON
C.2 Grammar of the Quartz Language

49  | PAUSE SEMICOLON
50  | THROW id SEMICOLON
51  | LES expr ORT SEMICOLON
52  | id COLON HALT SEMICOLON
53  | HALT SEMICOLON
54  | id COLON DEL1 expr_list DEL2 SEMICOLON
55  | ID DEL1 expr_list DEL2 SEMICOLON
56  | id COLON SUSTAIN DEL1 id DEL2 SEMICOLON
57  | SUSTAIN DEL1 id DEL2 SEMICOLON
58  | id COLON WAIT dexprs
59  | WAIT dexprs
60  | id COLON WAIT IMMED dexprs
61  | WAIT IMMED dexprs
62  | id COLON WEAK SUSPEND Stmt WHEN IMMED dexprs
63  | WEAK SUSPEND Stmt WHEN IMMED dexprs
64  | id COLON SUSPEND Stmt WHEN IMMED dexprs
65  | SUSPEND Stmt WHEN IMMED dexprs
66  | SUSPEND Stmt WHEN dexprs
67  | WEAK ABORT Stmt WHEN IMMED dexprs
68  | WEAK ABORT Stmt WHEN dexprs
69  | ABORT Stmt WHEN IMMED dexprs
70  | ABORT Stmt WHEN IMMED dexprs
71  | DO Stmt WHILE dexprs
72  | DURING Stmt HOLDS dexprs
73  | LOOP Stmt ID COLON EACH dexprs
74  | LOOP Stmt EACH dexprs
75  | IF dexpr Stmt
76  | IF dexpr Stmt ELSE Stmt
77  | SWITCH case_list ELSE Stmt
78  | CHOOSE Stmt ELSE Stmt
79  | LET DEL1 id ASSIGN expr DEL2 Stmt
80  | GENSEQUENCE Generic Stmt
81  | GENSENCPAR Generic Stmt
82  | GENASYNCPAR Generic Stmt
83  | GENINTERLEAVE Generic Stmt
84  | INSTANCE Qtype id ASSIGN expr OF BASE Stmt RECURSION Stmt
85  | TRY DDEL1 id DDEL2 Stmt
86  | ABSTRACT Stmt
87  | WHILE dexpr Stmt
88  | LOOP Stmt
89  | DEL1 id COMMA id DEL2 COLON EVERY dexpr Stmt
90  | EVERY dexpr Stmt
91  | Stmt SYNCPAR Stmt
92  | Stmt OR Stmt
93  | Stmt ASYNCPAR Stmt
94  | Stmt OR Stmt
95  | Stmt OR Stmt
96  | BDEL1 LocStmt BDEL2

97  | fpeq_list :
98        | MU id ASSIGN expr SEMICOLON fpeq_list
99        | NU id ASSIGN expr SEMICOLON fpeq_list

100 expr_list :
101   | expr
102   | expr COMMA expr_list

103 arr_expr : id DDEL1 expr DDEL2
104   | arr_expr DDEL1 expr DDEL2

105 expr : id
106   | TRUE
107   | FALSE
108   | INTCONST
109   | NATCONST
110   | BVCONST
111   | OCTCONST
426  C Concrete Grammar of the Quartz Language

112  | HEXCONST
113  | arr_expr
114  | CASTINT expr
115  | CASTNAT expr
116  | CASTBV expr
117  | DEL1 expr QUESTIONMARK expr COLON expr DEL2
118  | expr EQQ expr
119  | expr IMP expr
120  | expr OR expr
121  | expr XOR expr
122  | expr AND expr
123  | NOT expr
124  | expr LES expr
125  | expr LEQ expr
126  | expr OR expr
127  | expr GEN expr
128  | expr RQQ expr
129  | expr NEQ expr
130  | expr MINUS expr
131  | expr PLUS expr
132  | expr MOD expr
133  | expr DIV expr
134  | expr MUL expr
135  | MINUS expr
136  | FORALL Generic expr
137  | EXISTS Generic expr
138  | LISTSUM Generic expr
139  | EXP2 DEL1 expr DEL2
140  | LOG2 DEL1 expr DEL2
141  | LENGTHOF DEL1 id DEL2
142  | SIZEOF DEL1 expr DEL2
143  | MININT DEL1 expr DEL2
144  | MAXINT DEL1 expr DEL2
145  | MAXNAT DEL1 expr DEL2
146  | REVERSE DEL1 expr DEL2
147  | DEL1 expr TWOCOLON expr DEL2
148  | expr APPEND expr
149  | expr BDEL1 expr COLON expr BDEL2
150  | expr BDEL1 expr COLON expr BDEL2
151  | expr BDEL1 expr COLON expr BDEL2
152  | expr BDEL1 expr COLON expr BDEL2
153  | DDEL1 expr PSTRONGUNITIL expr DDEL2
154  | DDEL1 expr PSTRONGBEFORE expr DDEL2
155  | DDEL1 expr PSTRONGWHEN expr DDEL2
156  | DDEL1 expr PWEAKUNITIL expr DDEL2
157  | DDEL1 expr PWEAKBEFORE expr DDEL2
158  | DDEL1 expr PWEAKWHEN expr DDEL2
159  | DDEL1 expr PSTRONGUNITIL expr DDEL2
160  | DDEL1 expr PSTRONGBEFORE expr DDEL2
161  | DDEL1 expr PSTRONGWHEN expr DDEL2
162  | DDEL1 expr PWEAKUNITIL expr DDEL2
163  | DDEL1 expr PWEAKBEFORE expr DDEL2
164  | DDEL1 expr PWEAKWHEN expr DDEL2
165  | FNEXT expr
166  | FALWAYS expr
167  | FEVENTUAL expr
168  | PALWAYS expr
169  | PEVENTUAL expr
170  | PSTRONGNEXT expr
171  | PWEAKNEXT expr
172  | EXISTSPATH expr
173  | FORALLPATH expr
174  | EI expr
175  | EC expr
176  | EQ expr
177  | EI1 expr
178  | E1C expr
C.3 Some Example Programs

179 | E1Q expr
180 | AI expr
181 | AC expr
182 | AQ expr
183 | AX expr
184 | EX expr
185 | AY expr
186 | EY expr
187 | MU id POINT expr
188 | NU id POINT expr
189 | FIXPOINTS fpeq_list IN expr
190 | FORALL id COLON Qtype POINT expr
191 | EXISTS id COLON Qtype POINT expr
192 | TO expr END
193 | TO expr OVER expr END
194 | FROM expr TO expr END
195 | FROM expr TO expr OVER expr END
196 | DEL1 expr DEL2
References


[38] BERRY, G. Bottom-up computation of recursive programs. *Informatique Théorique et Applications (ITA)* 10, 1 (March 1976), 47–82.


432 References


References 435


References


[208] PLOTKIN, G. LCF considered as a programming language. Theoretical Computer Science 5, 3 (December 1977), 223–255.


References 441


Index

[100], 388, 434  
[101], 357, 385, 434  
[102], 388, 434  
[103], 211, 239, 241, 434  
[104], 34, 435  
[105], 215, 435  
[106], 241, 435  
[107], 9, 215, 241, 251, 252, 257–259, 435  
[108], 241, 435  
[109], 4, 435  
[110], 215, 429  
[111], 31, 357, 435  
[112], 11, 435  
[113], 212, 435  
[114], 215, 435  
[115], 359, 435  
[116], 359, 435  
[117], 241, 435  
[118], 9, 215, 252, 435  
[119], 215, 241, 252, 435  
[120], 188, 215, 429  
[121], 9, 215, 241, 252, 435  
[122], 3, 436  
[123], 216, 436  
[124], 87, 211, 239, 241, 436  
[125], 3, 436  
[126], 3, 436  
[127], 226, 230, 436  
[128], 224, 436  
[129], 357, 436  

[12], 215, 429  
[130], 4, 436  
[131], 4, 436  
[132], 4, 436  
[133], 4, 436  
[134], 44, 436  
[135], 212, 436  
[136], 215, 436  
[137], 224, 436  
[138], 5, 7, 212, 215, 436  
[139], 5, 7, 212, 215, 436  
[140], 373, 437  
[141], 5, 437  
[142], 212, 437  
[143], 8, 214, 437  
[144], 214, 437  
[145], 224, 437  
[146], 357, 381, 383, 385, 387, 437  
[147], 8, 214, 437  
[148], 357, 437  
[149], 11, 437  
[150], 248, 437  
[151], 388, 437  
[152], 34, 437  
[153], 346, 347, 437  
[154], 9, 437  
[155], 241, 437  
[156], 9, 215, 241, 251, 252, 257–259, 437  
[157], 215, 241, 251, 252, 257, 259, 437  
[158], 280, 284, 285, 293, 437
446 Index

[158], 215, 437
[159], 211, 214, 242, 437
[15], 215, 430
[160], 241, 438
[161], 241, 438
[162], 8, 214, 242, 438
[163], 8, 214, 242, 438
[164], 5, 7, 211, 214, 242, 438
[165], 4, 438
[166], 4, 438
[167], 44, 438
[168], 44, 438
[169], 72, 438
[170], 188, 430
[171], 72, 438
[172], 214, 438
[173], 34, 438
[174], 333, 438
[175], 21, 232, 438
[176], 2, 438
[177], 213, 438
[178], 212, 215, 239, 240, 438
[179], 4, 438
[17], 188, 215, 430
[180], 83, 219, 438
[181], 212, 246, 247, 439
[182], 241, 245, 439
[183], 230, 439
[184], 215, 439
[185], 388, 439
[186], 388, 439
[187], 1, 439
[188], 212, 213, 215, 239, 439
[189], 7, 8, 212, 213, 215, 237, 239, 240, 439
[18], 215, 430
[190], 216, 439
[191], 439
[192], 214, 439
[193], 215, 439
[194], 215, 439
[195], 388, 439
[196], 248, 439
[197], 196, 439
[198], 9, 211, 439
[199], 8, 214, 248, 440
[19], 388, 430
[1], 329, 429
[200], 231, 232, 440
[201], 357, 358, 440
[202], 357, 440
[203], 212, 440
[204], 212, 440
[205], 211, 241, 440
[206], 211, 241, 440
[207], 224, 440
[208], 212, 213, 215, 239, 240, 440
[209], 83, 440
[210], 388, 430
[211], 146, 440
[212], 286, 287, 440
[213], 286, 287, 292, 440
[214], 241, 440
[215], 385, 440
[216], 212, 440
[217], 388, 440
[218], 231, 441
[219], 329, 333, 334, 338, 339, 441
[220], 21, 224, 430
[221], 215, 441
[222], 145, 441
[223], 85, 92, 107, 145, 325, 441
[224], 92, 107, 129, 130, 145, 441
[225], 34, 39, 342, 441
[226], 61, 137, 441
[227], 145, 161, 441
[228], 145, 161, 441
[229], 61, 85, 98, 142, 143, 156, 441
[230], 61, 85, 156, 441
[231], 195, 441
[232], 195, 441
[233], 145, 146, 296, 298, 300, 441
[234], 388, 442
[235], 373, 442
[236], 212, 442
[237], 44, 442
[238], 215, 442
[239], 215, 442
[240], 8, 214, 430
[241], 7, 214, 442
[242], 214, 442
[243], 7, 8, 214, 442
[244], 316, 442
Index 447

[245], 262, 273, 280, 284, 285, 442
[246], 262, 273, 280, 284, 285, 442
[247], 297, 309, 442
[248], 89, 442
[249], 297, 309, 442
[24], 215, 430
[250], 297, 309, 442
[251], 88, 346, 347, 442
[252], 224, 443
[253], 83, 89, 443
[254], 87, 194, 211, 241, 443
[255], 8, 214, 443
[256], 8, 443
[257], 215, 443
[258], 7, 213, 240, 443
[259], 215, 443
[25], 44, 430
[260], 214, 443
[261], 215, 443
[262], 8, 248, 443
[263], 443
[264], 388, 443
[265], 316, 443
[266], 215, 443
[26], 341, 430
[27], 262, 271, 280, 293, 430
[28], 9, 430
[29], 241, 430
[2], 215, 429
[30], 262, 271, 277, 280, 283, 430
[31], 283, 285, 430
[32], 262, 271, 280, 292, 430
[33], 262, 271, 280, 431
[34], 9, 431
[35], 241, 251, 431
[36], 9, 215, 251, 252, 257–259, 431
[37], 215, 251, 252, 257–259, 431
[38], 8, 431
[39], 8, 212, 213, 215, 239, 240, 431
[40], 215, 429
[41], 108, 128, 431
[42], 9, 431
[43], 10, 11, 61, 77, 83, 89, 108, 128, 146, 147, 189, 431
[44], 85, 317, 431
[44], 213, 240, 431
[45], 9, 77, 431
[46], 8, 188, 242, 431
[47], 241, 242, 245, 431
[48], 8, 245, 431
[49], 8, 214, 248, 431
[50], 215, 432
[51], 259, 432
[52], 259, 432
[53], 61, 143, 432
[54], 146, 432
[55], 8, 214, 432
[56], 212, 213, 240, 432
[57], 85, 189, 432
[58], 9, 214, 251, 432
[59], 9, 214, 251, 432
[60], 214, 429
[61], 357, 432
[62], 215, 432
[63], 215, 432
[64], 98, 156, 432
[65], 215, 432
[66], 241, 432
[67], 4, 432
[68], 116, 433
[69], 227, 433
[70], 39, 433
[71], 212, 433
[72], 215, 433
[73], 329, 433
[74], 212, 433
[75], 338, 437, 433
[76], 215, 433
[77], 214, 433
[78], 357–359, 433
[79], 215, 433
[80], 215, 433
[81], 215, 433
[82], 215, 433
[83], 215, 433
[84], 215, 433
[85], 8, 248, 433
[86], 224, 433
[87], 224, 434
[88], 224, 434
[89], 434–48, 434
[90], 246, 247, 434
[91], 286, 434
abortion statements, 55
abstract interpretation, 49
abstraction, 49
action statement, 49
acyclic programs, 185
addition of radix-B numbers, 367
arithmetic operations on integers, 389
arithmetic operators, 359
assertion statement, 51
assignment, 50
assumption statement, 51
asynchronous circuit, 136
asynchronous parallel statement, 54, 73
atomic data type, 13
B-complement numbers, 391
behavioral modules, 43
Bekič’s lemma, 341
carrier, 180
causality analysis, 97
causality problem, 61
choose statement, 52, 73
combinational cycles, 185
comparison of radix-B numbers, 368
complement of radix-B numbers, 369
complete partial order, 333
composite data type, 13
compositional reasoning, 49
compound statements, 52
conditional statement, 52
conjunctively active parallel statements, 54, 70
continuous function, 333
conversion to radix-B numbers, 366
core statement, 67
data streams, 43
declaration, 45
depth, 86
digit extension lemma, 392
directed set, 333
directed sets, 331
disjunctively active parallel statements, 54, 70
division of B-complement numbers, 403
division of radix-B numbers, 375
division theorem for integers, 389
do-while loop, 54
during statement, 56
dynamic typing, 32
environment, 86
environment, partial order on, 88
every statement, 55
fixpoint theorem, 347
greatest lower bound, 329
higher radix representation, 365
immediate abortion, 55
immediate suspension, 55
infimum, 329
information flow, 13
instantaneous statement, 49
integer axioms, 389
interface, 45
interleaving parallel statement, 54, 73
Karatsuba multiplication, 373
lattice, 343
lazy evaluation, 87, 131
least upper bound, 329
let-abbreviation, 56
local variable declaration, 56
loop-each statement, 55
macro statement, 67
MaxInt, 392
MaxNat, 364
MinInt, 392
module instantiation, 56
monotone function, 333
multiplication of radix-B numbers, 372
must/cannot analysis, 97
non-restoring division, 414
nondeterministic statements, 73
parallel statement, 52
partial order, 329
path formulas, 37
path quantifiers, 34
Peano Axioms, 358
preemption statements, 56
primitive recursive definitions for natural numbers, 359
quotient digit estimation, 381
radix-B numbers, 363
reaction to absence, 124, 180
reincarnations, 58
restoring division, 413
runtime errors, 32
schizophrenic statements, 58
scope, 45
sequential statement, 52
simulation relation, 49
simultaneous fixpoints, 341, 342
SOS reaction rules, 84
SOS transition rules, 84
specification modules, 43
speed independence, 146
state formulas, 37
static expression, 14
static typing, 32
storage class, 45
storage type, 13
streams, 43
strict function, 87
strong abortion, 55
subtraction of radix-B numbers, 367
subtraction of radix-B numbers by adding the complement, 370
supremum, 329
surface, 86
suspension statement, 55
switch statement, 52
synchronous parallel statement, 52
Tarski/Knaster fixpoint theorem, 347
ternary simulation, 146
unary number system, 359
variable assignment, 86
weak abortion, 55
weak suspension, 56
while loop, 54
worst case reaction time, 44