1. Introduction

Formal verification is the task to find mathematical proofs that ensure that a given specification holds for a certain design. Although more powerful than traditional simulation, a breakthrough in the industrial use has been achieved only after the introduction of binary decision diagrams and symbolic state traversal algorithms [Brya86, CoBM89a], avoiding the explicit enumeration of states of a finite state machine model. These methods lead to powerful techniques like symbolic model checking [BCMD92], symbolic trajectory evaluation or language containment based on $\omega$-automata (the latter two can be found in this book on the pages 3 and 206, respectively). Using these approaches, a fully automated verification of significantly large systems has become possible. Common to all of these automated approaches is the view that the system to be verified has a finite number of states such that the verification problem is decidable. However, the drawback of these formalisms is that the size of the systems that can be verified with acceptable resources (in terms of memory and runtime) is limited due to the so-called state explosion problem that arises in most cases when the system has a nontrivial data path.

For this reason, the verification of data paths is often not done by these methods. As the data structures that are used in a system can in most cases be defined inductively, the verification is usually done by induction based methods. One can distinguish between explicit induction approaches [Burs69, Aubi79, BoMo79, GaGu89] and inductionless induction methods [Muss80, HuHu82, Frib86, JoKo86, JoKo89, BoRu93]. The latter ones are closely related to term rewrite methods [Ders90] and work more or less automatically (guided by some user interactions). However, they can only be applied to verification goals that are given as equation systems, possibly extended by conditional equations. Explicit induction can be applied to a broader class of goals, but requires a higher degree of manual interaction, as e.g. the induction variables, the induction schema and the induction hypotheses have to be chosen manually.

Hence, almost all formalisms used for formal verification can be classified into two categories: on the one hand there are decidable formalisms, unable to

* This work has been funded by a German research grant (Deutsche Forschungsgemeinschaft, SFB 358, TP C2.)
reason about abstract data types\(^1\), and on the other hand there are undecidable formalisms, able to reason about recursive data type by induction proofs. While these two classes of formalisms are limited to either control or data path dominated systems, this restriction does not hold for theorem provers that are based on higher-order logic [Gord86, ORSS94]. Unfortunately, these approaches require a considerable amount of manual interaction. Thus various approaches have been presented to partially automate the verification by incorporating automated reasoning procedures [KuSK93a, ORSS94] or by adding abstraction and compositional verification techniques to allow larger systems to be verified by finite state approaches (see [CILM89a, ClGL92, Long93, Schn96a] or many of the other approaches presented in this book).

Clearly, it is useful to separate control and data path oriented parts of a verification goal prior to verification [ScKK94a, LaCe94, HoBr95, HuGD95, ClZh95]. Proceeding this way, finite state approaches may be used for the controller part 'guiding' the verification of the data path, whereas the latter often requires theorem proving techniques. However, these approaches are also not able to provide full automation for systems with heavy interaction between control and data.

In this chapter, a new approach is presented that allows the combination of different proof techniques. These proof procedures and the methods to combine them have been implemented in the verification system C@S\(^2\) of the University of Karlsruhe. C@S is based on a linear temporal predicate logic that is a superset of both linear temporal (propositional) logic [Pnue77] and first order predicate logic with (inductive) abstract data types. To be precise, the underlying formalism of C@S is a higher order logic as the induction principles of the abstract data types are not expressible in first order predicate logic. However, the logic is far less powerful than the higher order logics used in HOL [GoMe93] or PVS [ORSS94]. Based on our experiments, we claim however that the logic is powerful enough to express almost all hardware verification problems and due to its special form of specifications, it allows a degree of automation that can not be reached by a general higher order theorem prover. In case of pure control-based systems, a full automation can be achieved as C@S provides state-of-the-art model checking procedures for temporal logic. Also, if a pure data path problem is to be verified, C@S provides induction methods that can be invoked to prove these problems semiautomatically, i.e. by giving some information in form of lemmata and term orderings, that can be seen as 'milestones' of the proof.

The integration of various decision procedure in interactive theorem provers is not a new idea. E.g. also the idea of the PVS system [ORSS94] is to obtain a more powerful higher order theorem prover by extending the

---

\(^1\) Reasoning about abstract data types leads to undecidable problems due to Gödel's theorem [MaYo78]. Hence, each method that is able to reason about abstract data types can no longer work fully automatically.

\(^2\) speak: cats
interactive prover by decision procedures, and the HOL system is also currently extended by automated proof procedures. The added value of COS is however, that these decision procedures do not only work isolated of each other. Instead, due to the restriction of the underlying logic, COS provides some special proof tactics that can be applied to split general proof goals such that finally various decision procedures can be used together for the remaining proof. Currently, COS has interfaces to the SMV [McMi93a] model checker implemented at the Carnegie Mellon University by Clarke et al. and the rewrite rule laboratory RRL [KaZh88] implemented by Kapur and Zhang. Furthermore, we have implemented a front end for SMV that enables SMV to check not only CTL formulas, but also to check and to prove LTL formulas (see section 5.3). As a result, COS currently offers the following decision procedures and proof methods:

1. decision procedures for verifying temporal properties (control dominated systems):
   a) CTL model checking
   b) linear time temporal logic (LTL) theorem proving and model checking
   c) \( \omega \)-automata
2. semi-automated proof procedures for proving lemmata about abstract data types
   a) explicit induction like cover set induction as implemented in RRL
   b) inductionless induction as implemented in RRL
3. interactive proof rules as e.g. invariant rules for breaking up interactions between control and data paths

Additionally, we are currently implementing additional proof procedures like an arithmetic decision procedures for Pressburger arithmetic and Büchi's monadic second-order theory of one successor. These arithmetic decision procedures can be used for proving lemmata that can be used for data path verification or for the verification of temporal properties.

In case of temporal properties, we suggest to use LTL instead of branching time temporal logic such as CTL for two reasons: Firstly, LTL temporal specifications can directly express facts where chains of events have to be considered, and secondly, LTL directly supports hierarchical reasoning in contrast to CTL in form of the assume-guarantee paradigm. The drawback of LTL is its principal non-linear time complexity for model checking and the insufficient support for efficiently implementable proof algorithms as compared to well-established symbolic CTL model checking implementations [BCMD90a]. However, the theoretical disadvantage of LTL does not necessarily lead to much larger runtimes in practice. Our approach contains new and efficient methods for LTL theorem proving and model checking. They are essentially based on transforming LTL verification problems into finite-state machine problems that can finally be checked by an arbitrary CTL model
checker as SMV. Proceeding this way, we use ω-automata as an intermediate formalism. Of course, it is also possible to use these automata directly for specifying temporal behavior. In our opinion, there are some examples as e.g. protocols, where it is more natural to use automata based specifications than temporal logics. This has also the advantage to directly visualize the specification as a finite-state transition system.

On the other hand, if a module is data-driven and does contain only a simple control flow, e.g. systolic architectures or signal processors, then neither temporal logics nor ω-automata are well suited to solve the problem due to their lack of reasoning about data types other than booleans. Based on our experiments, we claim that in most cases a first-order formalization together with possibilities to perform inductive proofs is sufficient. Moreover, data paths can be well described by data-assigning equations; hence, the best-suited proof methods are term rewrite methods together with inductionless induction capabilities. In many cases, only a few interactions are necessary and even completely automatic proofs can be obtained for some systems.

If a system contains both a complex control and a data path that interact with each other, then neither temporal logics nor first order predicate logics are well suited for performing hardware verification. In such cases, COS offers two solutions: if the data path is small, abstract data types are transformed to a purely propositional level. Then temporal logics or ω-automata are used to automatically verify the system. Obviously, this fails for the well-known problems, i.e. if the data path is too large (in terms of resulting propositional variables). Moreover, this approach does also not allow to reason about generic modules such as n-bit components. In these cases, COS offers methods to eliminate the temporal part of the specification: it is often possible to translate the temporal logic part into an ω-automaton with only safety properties (see section 3.2). Such automata can be interpreted directly as rewrite systems, and can hence be tackled by term rewrite methods and inductionless induction techniques, even when they are extended by abstract data types. If such a translation is not possible, then COS offers invariant rules for each temporal operator. These invariant rules allow to eliminate a temporal operator such that the remaining temporal part can be translated into an ω-automaton with safety properties, i.e. into a rewrite system.

Moreover, the possibility of interactive rule applications in COS allows to give the system 'proof hints' that also may significantly reduce the effort of a model checking proof. Proceeding this way it is sometimes possible to verify a system with a large bit width by model checking techniques, not being possible before. A simple, but powerful algorithm for heuristically choosing the suitable combination of proof procedures for a given system leads to a further automation.

There are two ways to read this article: on the one hand, the reader who is interested in the theory and the algorithms used in COS will find in section 3 the basic formalisms, i.e. the specification language of COS and in section
the new proof procedures of COS. For example, we explain in detail how linear temporal logic is translated into special $\omega$-automata. A reader more interested in practical applications will find in section 4 the definition of our system description language and in section 6 many case studies showing which verification procedure(s) of COS are used to verify which circuit classes and specifications.

2. Related Work

2.1 Hardware Verification using Temporal Logics

In principle, for each logic $\mathcal{L}$, there is a temporal extension $\mathcal{L}_t$ of this logic $\mathcal{L}$, where the truth value of a formula depends on time. The semantics of the temporal logic $\mathcal{L}_t$ is given by a sequence $M_0, M_1, \ldots$ of models for $\mathcal{L}$, where it is assumed that $M_i$ contains the truth values at time $i$ and that each transition from one model $M_i$ to the next one $M_{i+1}$ in the sequence requires one unit of time. The syntax of $\mathcal{L}$ is extended by temporal operators [Emer90] that express temporal relations between formulas, as e.g. $G \varphi$ which means that $\varphi$ has to hold for all future models.

Using this extension for propositional logic, linear temporal logic (LTL) [Emer90] is obtained. Most temporal properties can be described in LTL in a very natural, concise and readable manner. Hence, LTL has been widely used for specifying temporal properties: Manna and Pnueli used LTL for specifying concurrent programs [MaPn81, MaPn82, MaPn83]; Bochmann was the first who used LTL for specifying hardware systems [Boch82]; Malachi and Owicki considered 'self-timed systems' [MaOw81] with LTL; Manna and Wolper used LTL for the description of the synchronization of communicating processes [MaWo84]. Lichtenstein and Pnueli developed an algorithm for checking a LTL formula $\varphi$ in a given finite state model $M$ with a runtime $O(||M||^2 2^{||\varphi||})$ [LiPn85], where $||\varphi||$ means the length of the formula and $||M||$ the number of states of $M$. Using this algorithm for LTL model checking, it is possible to check small LTL formulas in big models. Moreover, LTL is well-suited for hierarchical verification [Lamp83, Pnue84].

In branching time temporal logics (BTL) [Emer90], a model is not a simple sequence of models of the base logic $\mathcal{L}$. Instead, at each 'point of time', there may exist more than one succeeding model which is chosen arbitrarily to model nondeterminism. Consequently, models of BTL form a tree instead of a linear sequence, where each node contains a traditional model for the logic $\mathcal{L}$. This tree is sometimes called a computation tree and therefore branching time temporal logics are sometimes called computation tree logics. If a system is described that consists of only a finite number of states, then it is possible to redraw the computation tree as a graph with only a finite number of nodes. In terms of modal logics, which are the general framework of temporal logics, this graph is called a Kripke structure [Krip75]. As each path through the
Kripke structure is a sequence of models, Kripke structures can also serve as an encoding of a set of models for LTL. Thus, BTL can be viewed as an extension of LTL and the complexity of the proof procedures cannot be better than the corresponding ones for LTL. On the other hand, Emerson and Lei showed that model checking problems\(^3\) for branching time temporal logic have the same complexity as the corresponding linear ones [EmLe87].

A general branching time temporal propositional logic that corresponds to LTL is CTL\(^*\) [Emer90]. Unfortunately, both the model checking problem for LTL and CTL\(^*\) have a high complexity, i.e. they are both PSPACE-complete [Emer90]. For this reason, Clarke and Emerson [ClEm81] developed a restricted computation tree logic called CTL. CTL is not powerful enough to describe all facts that can be formalized in LTL or CTL\(^*\) [Lamp80, LiPn85]. The advantage of CTL is however the efficient solvability of the model checking problem: given a Kripke structure \(M\) and a CTL formula \(\varphi\), we can determine the sets of states in \(M\) where \(\varphi\) holds in time \(O(||M||^2 ||\varphi||)\), where \(||\varphi||\) means the length of the formula and \(||M||\) the number of states of \(M\) [CIES86]. Hence, CTL has been used for the verification of quite large systems as for example the alternating bit protocol [CIES86], the Gigamax cache coherence protocol [McMi93a], the Futurebus cache coherence protocol [CGHJ93a], a traffic light controller [BCDM86] or a DMA system [McMi93a]. A disadvantage of CTL is its weak expressiveness which does for example not allow to state that a property has to hold infinitely often. For this reason, CTL has been extended also by fairness constraints [EmLe85a, CIES86] leading to roughly the same complexity of the model checking problem (additionally, it is linear in the size of the fairness constraints), but that have still a limited expressiveness.

Another disadvantage of CTL is that it cannot be easily used for a hierarchical verification where the model is given as a composition of submodules which can be replaced by previously verified specifications. In order to use the given design hierarchy despite this fact, some approaches have been discussed: Clarke, Long and McMillan modeled in [CILM89a, CILM89b] the environment of a module by an interface process and stated conditions for the correctness of this approach. Grumberg and Long [GrLo91] developed an approach, which is based on an ordering of the models and allowed to state similar to [Pnue84] that temporal properties hold for all systems that are implemented by the considered component. However, this approach required a further restriction for CTL (the use of existential path quantifiers has to be forbidden) that leads to the temporal logic ACTL.

Apart from LTL and BTL, interval temporal logic (ITL) [HaMM83] has been proposed for the specification of temporal properties. In contrast to

\(^3\) Verification problems are often modeled in such a manner that the system is mapped on a logical model and the specification on a logical formula. Hence, the verification problem becomes a model checking problem, where the truth value of a formula (the specification) is to be checked in a given model (the system).
LTL or BTL, the truth value of a formula is related to a sequence of states rather than to a single state. The most interesting operator of ITL is the 'chop operator ;' which has the following effect: \( \phi ; \psi \) holds on a sequence of states iff this sequence can be split up in two parts, where \( \phi \) hold on the first and \( \psi \) on the second part. The chop operator can be used to describe many interesting facts in ITL. However, the satisfiability of ITL formulas is not decidable [HaMM83], and hence, the logic has been used only in some rare cases [Mosz85, Mosz85a, Lees88a] on transistor level.

2.2 Hardware Verification Based on Predicate Logics

Propositional logics and their temporal extensions are not well suited for describing systems that process abstract data types, as the data types must first be mapped to the propositional level. In order to use abstract data types directly in the logic, i.e. to have variables of the corresponding type, predicate logics and their temporal extensions have to be used instead. The research in this area is mainly based on the Boyer-Moore logic and its prover and also to approaches based on higher order logic.

The Boyer-Moore logic is a quantifier-free first order predicate logic with the extension to formalize induction schemes [BoMo88]. The logic as well as the corresponding interactive prover have been developed by R.S. Boyer and J.S. Moore. The prover allows to define new inductive data types by declaring constructor and destructor functions for this data type. Of course, it is also possible to define new functions on the defined data types. In order to avoid inconsistent definitions, these definitions have to fulfill certain constraints. For example, the recursive definitions must follow a well-ordering such that a termination of the recursion is guaranteed. The Boyer-Moore prover has strong heuristics, but does not work automatically. Usually, the user has to provide appropriate lemmata which are useful for the proof. Boyer-Moore has already been used for verifying hardware: Hunt verified in 1985 the FM8501 microprocessor [Hunt85] and 1989 the 32 bit extension FM8502 of FM8501 [Hunt89]. The verification used generic \( n \)-bit wide descriptions in order to make use of the induction tactics of the Boyer-Moore prover.

Apart from the explicit induction methods implemented in the Boyer-Moore prover, there are other induction methods which are summarized under the category 'inductionless induction' [Muss80, HuHu82, Frib86, JoKo86, JoKo89, BoRu93] and are available in theorem provers as e.g. the rewrite rule laboratory RRL of Kapur and Zang [KaZh88], the ReDuX system [Bouh94, BeBR96] or SPIKE [BoRu93]. Inductionless induction methods are based on term rewriting, which is a general theorem proving method. Examples, where term rewrite methods have been used for formal verification are [NaSt88], where the verification of of the Sobel chip has been presented, [KaSu96] where multipliers have been verified and [ChPC88], where term rewriting has been used for proving propositional formulas. Term rewriting is also available in almost every interactive proof assistant, in particular in
those for higher order logics. In contrast to explicit induction methods, the advantage of these methods is that they are able to generate their own induction hypothesis and hence, offer in principle a higher degree of automation than explicit induction methods. However, it is in most cases necessary to formulate some lemmata that are useful for the later proof. These lemmata are then automatically proved first, and by their help, the remaining proof can also be done automatically.

In comparison to (temporal) propositional logic, a major advantage of first-order logic in combination with induction schemes is the use of abstract data types. On the other hand, the temporal behavior of a system can not be modeled appropriately in this logic. For this reason, C@S is based on a temporal extension of second-order predicate logic with induction schemes, as outlined section 3. From a formal point of view, this logic is a higher order logic, but it is strictly weaker than usual higher order logics as supported by higher order proof assistants as e.g. HOL [GoMe93], VERITAS [HaDa86], NUPRL [Cons86], ISABELLE [Paul94], LAMBDA [FoHa89] and PVS [ORSS94].

A disadvantage of this logic is its undecidability and its incompleteness that holds for each higher order logic: each decidable set of rules is not powerful enough to derive all theorems [MaYo78]. In practice, this incompleteness is however not a serious drawback. Usually, the set of inference rules to be applied manually is sufficient to prove all verification tasks. The advantage of our logic is that it contains exactly the constructs that are necessary from our point of view for the specification of systems with a complex temporal behaviour together with a complex data processing.

2.3 Combined Approaches

The key idea of C@S is the combination of different formalisms for performing hardware proofs. Compositional reasoning and abstraction play also important parts in our approach. Both aspects will be briefly related to other existing approaches.

Using combined approaches for hardware verification can be motivated twofold: first, not all formalisms are equally well suited for expressing different properties and abstractions. Thus, as it has been shown in the last section, temporal logics are better suited for reasoning about time, whereas predicate logics are well suited for dealing with data paths. If however, both kind of properties are to be verified then both aspects have to be treated simultaneously. The second reason for combining approaches is to enrich an interactive proof tool with decision procedures thus that at least the decidable parts of the verification can be automated, although an undecidable framework is used.

The idea of enriching interactive theorem provers for higher order logic with powerful decision procedures can also be found in the PVS system. However, PVS does not support a proof methodology which is based on a separation of control and data flow and no common formalism as available in
COS is used. Our specification language is based on linear time temporal logic (LTL), enriched by abstract data types. This language is similar to the CTL extension used for word level model checking [ClZh95]. However, [ClZh95] aims at a full automation, hence only finite bitvectors are possible as abstract data types. The approach of Hojati and Brayton [HoBr95, HoDB97] can be seen as complementary to our approach as especially for systems without full control/data interaction valuable decidability results have been achieved, which can be directly used in COS. By using invariants we are also able to cope with those systems, not treated by them – the price to be paid for that is interaction.

The approach of Hungar, Grumberg and Damm [HuGD95] is also based on the idea to separate control and data. They however use FO-ACTL (first order version of ACTL), whereas we support full LTL enhanced by data equation expressions. As they use a first-order variant of temporal logics, the reasoning process has to be performed using a dedicated calculus involving the construction of first-order semantic tableaux. Instead, COS uses a more rigorous separation of control and data such that already existing proof tools for propositional temporal logics may be used in conjunction with e.g. term rewriting systems.

COS is a direct descendant of MEPHISTO [ScKK91a, ScKK91b, KuSK93a]. MEPHISTO was a verification system that was implemented by our group on top of the HOL system [GoMe93]. Similar to COS, MEPHISTO provided the user with tactics for structuring the proof goals such that a further automation of the proof had been achieved by a first order theorem prover called FAUST [ScKK92b, ScKK93a, ScKK94b]. A key idea of MEPHISTO was also to use so-called ‘hardware formulas’ as a restricted form for specifications in order to achieve a higher degree of automation of the proofs inside the HOL system. These hardware formulas have been extended in the COS system to the fair prefix formulas, and new algorithms for translating linear temporal logic to these fair prefix formulas have been developed and implemented in COS. COS is no longer on top of HOL, but a complete system on its own that has interfaces to include other verification systems such as SMV or RRL.

2.4 Outline of the Chapter

In the next section, the basics of the underlying formalism of COS are given. First, we define a second-order temporal logic that is the basis of the COS logic and a special kind of ω-automata called fair prefix automata. The strategy of COS is to split verification goals given in this logic such that they belong to a subset of the logic where automated proof procedures are available. In this book section, we mainly focus on two proof procedures of COS: one for translating linear temporal propositional logic to fair prefix automata, shown in detail in section 3.2, and a proof procedure for equation systems, given in section 3.3. It is important to note here, that fair prefix formulas that have only safety properties can be directly interpreted as rewrite systems.
and hence, can be proved on the one hand by model checking procedures and on the other hand by rewrite based methods. The latter proof method allows also the extension to abstract data types. If the translation of a LTL formula does not yield in such an \( \omega \)-automaton, then \( \text{COS} \) allows to modify the formula by interactive proof rules such that the modified formulas can finally be translated to a rewrite system.

3. Basics

The description of the entire behavior of general systems requires one hand to express temporal behavior and on the other hand to reason about abstract data types defined by the user. The logic used in \( \text{COS} \) is powerful enough to define new abstract data types and to express temporal behavior, but is on the other hand weak enough such that a high degree of proof automation can be achieved. For this reason, proof rules of \( \text{COS} \) aim at reducing formulas to subsets where powerful proof procedures are available: currently, these are on the one hand linear temporal propositional logic together with finite state \( \omega \)-automata and, on the other hand, rewrite systems with abstract data types. These logics are considered in detail in the following three subsections.

3.1 Underlying Formalism of \( \text{COS} \)

The underlying formalism of \( \text{COS} \) is a second order linear time temporal logic with inductively defined abstract data types. This logic is on the one hand a proper subset of higher order logic, and on the other hand a common extension of LTL and first order logic with abstract data types. The syntax of this logic is determined by a signature which fixes the function symbols, constants and variables.

Definition 3.1 (Signature). A signature \( \Sigma \) is a tuple \( (C_\Sigma, V_\Sigma, TP_\Sigma, \text{typ}_\Sigma) \), where \( C_\Sigma \) is the set of constant symbols, \( V_\Sigma \) is the set of variables, and \( TP_\Sigma \) is a set of basic types. \( \text{typ}_\Sigma \) is a function that assigns to symbols of \( C_\Sigma \cup V_\Sigma \) a type of the form \( \alpha_1 \times \ldots \times \alpha_n \rightarrow \beta \) where \( \alpha_i, \beta \in TP_\Sigma \). In case \( n = 0 \), simply a type \( \beta \in TP_\Sigma \) is assigned. It is assumed that \( TP_\Sigma \) contains at least the set of boolean values \( \mathbb{B} \).

Usually, we also have natural numbers \( \mathbb{N} \) in the signature, but these can also be defined as shown in the following. Given a signature that provides basic types and symbols for constants and variables, we can define the sets of terms and the set of formulas.

Definition 3.2 (Terms \( T_\Sigma \)). The set of terms \( T_\Sigma \) over a signature \( \Sigma = (C_\Sigma, V_\Sigma, TP_\Sigma, \text{typ}_\Sigma) \) is the smallest set that satisfies the following properties:

1. \( x \in T_\Sigma \) for all \( x \in V_\Sigma \)
2. \( f(\tau_1, \ldots, \tau_n) \in \mathcal{T}_\Sigma \) with \( \text{typ}_\Sigma(f(\tau_1, \ldots, \tau_n)) := \beta \) for all \( f \in \mathcal{C}_\Sigma \) provided that \( \text{typ}_\Sigma(f) = \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta \), \( \text{typ}_\Sigma(\tau_i) = \alpha_i \) and \( \tau_1, \ldots, \tau_n \in \mathcal{T}_\Sigma \).

3. Given that \( \tau \in \mathcal{T}_\Sigma \) with \( \text{typ}_\Sigma(\tau) = \alpha \), then \( \chi_\tau \in \mathcal{T}_\Sigma \) with \( \text{typ}_\Sigma(\chi_\tau) = \alpha \).

It has to be noted here, that the set of terms is defined as usually for first-order predicate logic (see e.g. [Fitt90]). However, we also have an additional terms \( \chi_\tau \) that represent the value of a term at the next point of time. These terms are necessary when temporal behavior is mixed with abstract data types. All other temporal operators work on formulas, i.e. of terms of type \( \mathbb{B} \).

**Definition 3.3 (Formulae \( \mathcal{T}_\Sigma \) of a signature).**

The set of formulas \( \mathcal{T}_\Sigma \) over a given signature \( \Sigma = (\mathcal{C}_\Sigma, V_\Sigma, TP_\Sigma, \text{typ}_\Sigma) \) is the smallest set that satisfies the following properties (all formulas have type \( \mathbb{B} \)):

1. \( q \in \mathcal{T}_\Sigma \) for all \( q \in V_\Sigma \) with \( \text{typ}_\Sigma(q) = \mathbb{B} \)
2. \( p(\tau_1, \ldots, \tau_n) \in \mathcal{T}_\Sigma \) for all \( p \in \mathcal{C}_\Sigma \) provided that \( \text{typ}_\Sigma(p) = \alpha_1 \times \cdots \times \alpha_n \rightarrow \mathbb{B} \), \( \text{typ}_\Sigma(\tau_i) = \alpha_i \) and \( \tau_1, \ldots, \tau_n \in \mathcal{T}_\Sigma \).
3. \( \tau_1 = \tau_2 \in \mathcal{T}_\Sigma \), for all terms \( \tau_1, \tau_2 \in \mathcal{T}_\Sigma \) with \( \text{typ}_\Sigma(\tau_1) = \text{typ}_\Sigma(\tau_2) \)
4. \( \neg \phi, \phi \land \psi, \phi \lor \psi, \phi \rightarrow \psi, \phi = \psi \in \mathcal{T}_\Sigma \) for all \( \phi, \psi \in \mathcal{T}_\Sigma \)
5. \( \chi_\phi, G\phi, F\phi, [\phi W \psi], [\phi W' \psi], [\phi U \psi], [\phi U W \psi], [\phi U U \psi] \in \mathcal{T}_\Sigma \), for all \( \phi, \psi \in \mathcal{T}_\Sigma \)
6. \( \exists x. \phi \in \mathcal{T}_\Sigma \) and \( \forall x. \phi \in \mathcal{T}_\Sigma \), for all \( x \in V_\Sigma \)
7. abtype \( \alpha \) with \( c_1, \ldots, c_n \) in \( \phi \in \mathcal{T}_\Sigma \) for all \( \phi \in \mathcal{T}_\Sigma \) provided that \( c_1, \ldots, c_n \in \mathcal{C}_\Sigma \) and \( \alpha \in TP_\Sigma \)

Moreover, let \( x = \tau \) in \( \phi \) end is used as syntactic sugar that corresponds to the formula that is obtained by replacing every occurrence of the variable \( x \) in \( \phi \) by \( \tau \).

The first three items introduce atomic formulas, while the fourth and fifth state the boolean closure and the closure under temporal operators, respectively. Item 6 introduces first and second order quantification. The second order quantification, in particular the quantification over variables of type \( \mathbb{B} \), is important for our formalism as it allows us to formalise finite automata in a very convenient way (see next subsection). The possibility to define new abstract data types is added by item 7 in the above definition, where the constants \( c_1, \ldots, c_n \in \mathcal{C}_\Sigma \) are defined as the constructors of the new data type \( \alpha \) according to [Melh88d]. The definition of \( \alpha \) is locally declared for the formula \( \phi \). We assume that the signature \( \Sigma \) has enough types for each user defined type definition.

Note that all constructs of the COS logic are viewed as depended on time, i.e. if we have a term of type \( \alpha \), then it is always assumed that the value of the term may change during time. Hence, it would be more precise to assign to the term the type \( \mathbb{N} \rightarrow \alpha \), but this would lead to an overhead of syntax
that can be avoided once we have made the convention that all types are interpreted as dependent on time.

After defining the syntax of the C@S logic, we define now its semantics. First, we need the notion of a variable assignment, interpretations and models.

**Definition 3.4 (Domains, Interpretations, Assignments).**
Given a signature \( \Sigma = (C_\Sigma, V_\Sigma, T_{P_\Sigma}, \text{typ}_\Sigma) \), each set \( D := \bigcup_{\alpha \in T_{P_\Sigma}} D_\alpha \) is called a domain for \( \Sigma \). A function \( I : C_\Sigma \rightarrow \bigcup_{\alpha \in T_{P_\Sigma}} D_\alpha \) that maps a constant of type \( \alpha \) to a function \( \mathbb{N} \rightarrow D_\alpha \) is called an interpretation for \( \Sigma \) and \( D \). Similarly, a function \( \xi : V_\Sigma \rightarrow \bigcup_{\alpha \in T_{P_\Sigma}} D_\alpha \) that maps a variable of type \( \alpha \) to a function \( \mathbb{N} \rightarrow D_\alpha \) is called a variable assignment for \( \Sigma \) and \( D \).

Given a signature, a domain, an interpretation and a variable assignment, we can define the semantics of arbitrary formulas and terms. This is done recursively on the structure of the corresponding term or formula. Note again, that each term and formula is mapped to a function that depends on time, where time is modeled by the natural numbers \( \mathbb{N} \). Functions are given in the following in a \( \lambda \)-calculus syntax, and the application of such a function \( f \) on a point of time \( t \) is written as \( f(t) \).

**Definition 3.5 (Semantics).** Given a signature \( \Sigma = (C_\Sigma, V_\Sigma, T_{P_\Sigma}, \text{typ}_\Sigma) \), and a domain \( D \) for \( \Sigma \), an interpretation \( I \) for \( \Sigma \) and \( D \), and a variable assignment \( \xi \) for \( \Sigma \) and \( D \), the semantics of terms and formulas is defined as follows:

1. \( \mathcal{V}_\xi^{D,I}(x) := \xi(x) \) for all \( x \in V_\Sigma \)
2. \( \mathcal{V}_\xi^{D,I}(f(t_1, \ldots, t_n)) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(f) \right]^{(t)}(\left[ \mathcal{V}_\xi^{D,I}(t_1) \right]^{(t)}, \ldots, \left[ \mathcal{V}_\xi^{D,I}(t_n) \right]^{(t)}) \)
3. \( \mathcal{V}_\xi^{D,I}(X\tau) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(\tau) \right]^{(t+1)} \)
4. \( \mathcal{V}_\xi^{D,I}(t_1 = t_2) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(t_1) \right]^{(t)} = \left[ \mathcal{V}_\xi^{D,I}(t_2) \right]^{(t)} \)
5. \( \mathcal{V}_\xi^{D,I}(\neg \varphi) := \lambda t. \neg \left[ \mathcal{V}_\xi^{D,I}(\varphi) \right]^{(t)} \),  
   \( \mathcal{V}_\xi^{D,I}(\varphi \land \psi) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(\varphi) \right]^{(t)} \land \left[ \mathcal{V}_\xi^{D,I}(\psi) \right]^{(t)} \),  
   \( \mathcal{V}_\xi^{D,I}(\varphi \lor \psi) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(\varphi) \right]^{(t)} \lor \left[ \mathcal{V}_\xi^{D,I}(\psi) \right]^{(t)} \)
6. \( \mathcal{V}_\xi^{D,I}(X\varphi) := \lambda t. \left[ \mathcal{V}_\xi^{D,I}(\varphi) \right]^{(t+1)} \),  
   \( \mathcal{V}_\xi^{D,I}(G\varphi) := \lambda t. \begin{cases} 1 & \mathcal{V}_\xi^{D,I}(\varphi)^{(t+\delta)} = 1 \text{ holds for all } \delta \in \mathbb{N} \\ 0 & \text{otherwise} \end{cases} \),  
   \( \mathcal{V}_\xi^{D,I}(F\varphi) := \lambda t. \begin{cases} 1 & \mathcal{V}_\xi^{D,I}(\varphi)^{(t+\delta)} = 1 \text{ holds at least for one } \delta \in \mathbb{N} \\ 0 & \text{otherwise} \end{cases} \)
The semantics of the type definitions assures that functions over the type \( \alpha \) can be defined recursively. This primitive recursion axiom assures also that the set \( D_{\alpha} \) is isomorphic to the set of variable-free terms that can be built from the constants \( c_1, \ldots, c_n \) and that the constructor functions \( c_i \) are one-to-one \cite{Melh88d}. This means on the one hand, that none of the terms \( c_i(x_1, \ldots, x_{m_i}) \) and \( c_j(y_1, \ldots, y_{m_j}) \) are mapped to the same element in \( D_{\alpha} \) and on the other hand, that for each element \( d \in D_{\alpha} \) there is a variable-free term \( c_i(x_1, \ldots, x_{m_i}) \) such that \( \nu_{\xi}^{D,I}(c_i(x_1, \ldots, x_{m_i})) = d \) holds.

As an example, consider the definition of the natural numbers:
abstype \mathbb{N} with 0 : \mathbb{N}, SUC : \mathbb{N} \rightarrow \mathbb{N} in \varphi

Here, we have type(0) = \mathbb{N} and type(SUC) = \mathbb{N} \rightarrow \mathbb{N}. The definition of the semantics now assures that Peano's axioms hold, i.e. for each element \( d \in D_\mathbb{N} \), we have a term of the form \( \text{SUC}^n(0) \) such that \( \forall^D \varphi \text{SUC}^n(0) = d \) holds, and of course the induction principle holds for \( \mathbb{N} \). This axiomatic definition of the new data types introduces directly an induction scheme that allows to reason inductively on the new data type. For more details, see section 3.3.

Clearly, also linear temporal propositional logic is a subset of the above defined logic. This logic is obtained when the set of basic types \( \mathcal{TP}_\Sigma \) of the signature \( \Sigma \) contains only the type \( \mathbb{B} \) and neither quantification over variables nor the definition of abstract data types is allowed. In the following, we consider two other important subsets of \( \mathcal{TL}_\Sigma \) which are important for specifying hardware modules: fair prefix formulas and first order abstract data types.

### 3.2 Fair Prefix Automata

In general, an \( \omega \)-automaton [Thom90a] is a finite state system for accepting infinite words according to an acceptance condition that is usually a simple LTL formula. For example, Büchi automata use as acceptance condition formulas of the form \( GF\varphi \) while Rabin automata use formulas of the form \( \bigwedge_{i=0}^I GF\varphi_i \lor FG\psi_i \), where \( \varphi, \varphi_i, \) and \( \psi_i \) are propositional formulas.

An important proof strategy of the C@S system is to translate the temporal logic part of a verification goal into an \( \omega \)-automaton that has an acceptance condition of the form \( G\varphi \) where \( \varphi \) is a propositional formula. As already mentioned, these automata can also be viewed as equation systems and can hence be simply proved with term rewriting methods hence they allow to verify temporal properties with abstract data types. The kind of \( \omega \)-automata that is considered in this section is strongly related to the class of \( \omega \)-regular languages which can be viewed as the boolean closure of safety (resp. liveness) properties [StWa74]. In order to capture full LTL, we extend these automata by additional fairness constraints.

Throughout this chapter, we describe \( \omega \)-automata as logical formulas similar to [Sief70]. For this reason, suppose that the states and the input alphabet is encoded by boolean tuples. In the following, these tuples are written as vectors \( \vec{b} \) and for \( \vec{b} = (b_1, \ldots , b_n) \), the length of \( \vec{b} \) is defined as \( \|\vec{b}\| := n \). An infinite word \( \vec{t} \) over an alphabet \( \Sigma = \mathbb{B}^n \) can be modeled as a function from natural numbers \( \mathbb{N} \) to \( \mathbb{B}^n \), where \( \vec{t}(k) \in \Sigma \) is a boolean tuple that encodes the \( k \)-th symbol of the sequence \( \vec{t} \). Formulas that contain exclusively the free variables \( \vec{b} = (b_1, \ldots , b_n) \) are written as function applications \( \Phi(\vec{b}) \) on \( \vec{b} \).

In general, each \( \omega \)-automaton can be described by a logical formula of the form \( \exists \vec{q}. T(\vec{i}, \vec{q}) \land \Theta(\vec{i}, \vec{q}) \), where \( \vec{q} \) are the state variables, \( \vec{i} \) are the input variables, \( T(\vec{i}, \vec{q}) \) is the transition relation, and \( \Theta(\vec{i}, \vec{q}) \) is the acceptance
A deterministic transition relation can always be given as a conjunction of initialisation equations of the form $q_k = \omega_k$ with $\omega_k \in \{1, 0\}$ and transition equations of the form $G \left( Xq_k = \Omega_k(i, q) \right)$. Fair prefix formulas are now defined in this style:

**Definition 3.6 (Fair Prefix Formulas (FPF)).**

Let $\Omega_k(i, q)$ for $k \in \{0, \ldots, s\}$, $\xi_l(i, q)$ for $l \in \{0, \ldots, f\}$, and $\Phi_m(i, q)$ and $\Psi_m(i, q)$ for $m \in \{0, \ldots, a\}$ be propositional formulas with the free variables $i$ and $q$. Moreover, let $\omega_k \in \{1, 0\}$ for $k \in \{0, \ldots, s\}$, then the following formula $P(i)$ is a (fair) prefix formula:

$$P(i) := \left( \forall q_1 \ldots q_s. \left( \bigwedge_{k=0}^s \left[ (q_k = \omega_k) \land G \left( Xq_k = \Omega_k(i, q) \right) \right] \land \left( \bigwedge_{l=0}^f GF \xi_l(i, q) \right) \rightarrow \left( \bigwedge_{m=0}^a [G \Phi_m(i, q)] \lor [F \Psi_m(i, q)] \right) \right) \right)$$

The formulas $\Phi_m(i, q)$ and $\Psi_m(i, q)$ are called the safety properties and liveness properties, respectively. $\xi_l(i, q)$ is called a fairness constraint.

Note that the definition includes that each FPF is deterministic and that each propositional formula over the variables $i$ and $q$ is the characteristic function of a set of edges in the state transition diagram of the automaton. The run $q$ of an input sequence $i$ is fair with respect to a set of edges $\xi_l(i, q)$, iff the run traverses infinitely often at least one of the edges of $\xi_l(i, q)$. An infinite word $i$ is accepted by a FPF $P(i)$, iff the corresponding run $q$ is fair with respect to all fairness constraints $\xi_l(i, q)$ and satisfies all acceptance pairs $(\Phi_m(i, q), \Psi_m(i, q))$ of $P(i)$ as follows: either it stays exclusively in the set $\Phi_m(i, q)$ or it reaches at least once an edge of $\Psi_m(i, q)$.

Languages that can be accepted by a FPF are exactly the languages that can be accepted by a deterministic Büchi automaton. We are however mainly interested in simple prefix formulas, i.e. FPFs without fairness constraints. In contrast to FPFs and deterministic Büchi automata, simple prefix formulas are closed under boolean operations. This is important for the translation procedure given in section 5.3.

### 3.3 Rewrite-Based Theorem Proving for Abstract Data Types

Formulae ‘abstype $\alpha$ with $c_1, \ldots, c_n$ in $\varphi$’ of the CÆS logic define $\alpha$ to be a type that is isomorphic to the set of ground terms that can be obtained by the constructor functions $c_1, \ldots, c_n$. Most abstract data types can be described in this way; for example, consider again the definition of the natural numbers:

abstype $\mathbb{N}$ with $0 : \mathbb{N}, SUC : \mathbb{N} \rightarrow \mathbb{N}$ in $\varphi$
The \( \text{COS} \) System is now a type visible in \( \varphi \) that is isomorphic to the set 0, \( \text{SUC}(0) \), \( \text{SUC}(\text{SUC}(0)) \), \ldots and new operations can be defined on \( \mathbb{N} \) by primitive recursion over the constructors 0 and \( \text{SUC} \). As another example, consider `abstype Nat_List with NIL : Nat_List, CONS : \mathbb{N} \times Nat_List \rightarrow Nat_List` in \( \varphi_\ell \). In this formula, a type of the name `Nat_List` that is isomorphic to the set of lists containing natural numbers is declared for the formula \( \varphi \).

Of course, it is possible to define also new operations on this new data type. Usually, \( \varphi \) is an implication, where the premise contains such definitions in form of a recursive definition over the constructors \( c_1, \ldots, c_n \). For example, \( \varphi_\ell \) could have the following form:

\[
\left( \text{TL (NIL)} = \text{NIL} \land \forall x \ y. \text{TL (CONS)}(x, y) = y \right) \rightarrow \varphi_1
\]

The premise defines a new function \( \text{TL} \) that removes the first element of a list if the list is not empty. \( \text{TL} \) can be used with this semantics in the remaining formula \( \varphi_1 \).

The reason for this form of defining new data types is that there are special proof methods for these data types that belong to the class of 'induction-less induction' proof methods [Muss80, HuHu82, Frib86, JoKo86, JoKo89, BoRu93]. Of course, the special form of definition directly leads to an induction scheme as we know all members of the set \( D_\alpha \) (these are the variable free terms built from the constructor constants of the data type). In the given examples, the induction schemes are as follows:

\[
\begin{align*}
P(0) & \forall x : \mathbb{N}. P(x) \rightarrow P(\text{SUC}(x)) \\
& \forall x : \mathbb{N}. P(x) \\
\text{P(NIL)} & \forall x : \mathbb{N} \ y : \text{Nat_List}. P(x) \rightarrow P(\text{CONS}(x, y)) \\
& \forall x : \text{Nat_List}. P(x)
\end{align*}
\]

The advantage of inductionless induction methods is that they are able to directly use and derive complex induction schemes from the definition of the abstract data types similar to the ones given above. The methods are also able to automatically generate intermediate lemmata for making a complex proof. However, in practice it is usually necessary to guide the proof search by providing the prover by some lemmata that one expects to be of some help for the current proof. On the other hand, these methods are limited to oriented equation systems, to be more precise to term rewrite systems [Ders90].

**Definition 3.7 (Term Rewrite System).** A term rewrite system is a set of equations of the form \( l = r \), where each equation is also called a rewrite rule. A rule \( l = r \) can be applied to a term \( t \) if this term \( t \) contains a subterm \( \tau \) such that there is a substitution \( \sigma \) of the variables occurring in \( l \) such that \( \sigma(l) = \tau \). The application of the rule \( l = r \) on the term \( t \) is then performed by replacing some occurrences of the subterm \( \tau \) in \( t \) by \( \sigma(r) \).
A term rewrite system is called terminating iff for each term \( t \) there is no infinite number of rule applications. A term rewrite system is confluent iff all possible rule applications finally yield in the same term. Finally, a rewrite system is called canonical, iff the system is terminating and confluent.

Two rewrite systems \( R_1 = \{ l_{1,1} = r_{1,1}, \ldots, l_{n,1} = r_{n,1} \} \) and \( R_2 = \{ l_{1,2} = r_{1,2}, \ldots, l_{m,2} = r_{m,2} \} \) are equivalent iff \( \bigwedge_{i=1}^{n} l_{i,1} = r_{i,1} \) \( \bigwedge_{i=1}^{m} l_{i,2} = r_{i,2} \) holds.

If the rewrite system contains more than one rule, it is undecidable whether the system is terminating. For this reason, usually term orderings are used [Ders87] to assure the termination property. The idea is to order the set of terms by a well-founded ordering \( \succsim \) such that the lefthand sides of the rewrite rules are larger than the righthand sides. As each chain of terms must have a minimal term, it is clear that the corresponding rewrite system must be terminating.

If it is known that a rewrite system is terminating, the confluence property is decidable. However, rewrite systems are often not confluent, although there is an equivalent confluent rewrite system that can be computed by the completion algorithm due to Knuth and Bendix [KnBe70]. This algorithm is also the basis of the inductionless induction proof methods. The basic idea of these methods is the following theorem [Ders83]:

**Theorem 3.1.** Given a a canonical rewrite system \( R_1 \) and a set of equations \( E = \{ e_1, \ldots, e_n \} \). Let \( R_2 \) be the canonical rewrite system that has been obtained by applying the completion algorithm on \( R_1 \cup E \). Then \( R_1 \vdash e_i \) holds iff the set of irreducible variable free terms of \( R_1 \) is also the set of irreducible variable free terms of \( R_2 \).

The computation of the irreducible variable free terms of a rewrite system is however undecidable. For this reason, some more specialized algorithms have been developed. For our special form of data type definitions, the algorithm of Huet and Hullot [HuHu82] is of particular use. This algorithm is based on the fact that we know for each data type the set of constructors, that the primitive recursion theorem holds for these constructors (see the definition of the semantics of our logic).

A function \( f \in \Sigma_D \) is completely defined iff for all constructors \( c_1, \ldots, c_n \) there is a reduction rule \( f(c_1(\ldots), \ldots, c_n(\ldots)) = r \). If all function symbols are completely defined in a rewrite system \( R \), and \( R \) is canonical, and no lefthand side of any equation of \( R \) starts with a constructor, then the proof procedure of [HuHu82] is given in figure 3.1. The algorithm has the following properties:

**Theorem 3.2.** Let \( \succ \) be a term ordering \( \succ \) and \( E_1 \) be a rewrite system with completely defined function symbols. If \( \text{HuHu}(\{\}, E_1 \cup E_2, \succ) \) terminates with \( \text{PROOF} \), then \( E_1 \vdash E_2 \) holds. The thereby obtained rewrite system \( R \) does also follow the definition principle and is equivalent to \( E_1 \). If \( \text{HuHu}(\{\}, E_1 \cup \)
(* INPUT rewrite system R, set of equations E, term ordering > *)
(* OUTPUT PROOF, DISPROOF oder FAILURE *)
PROCEDURE HuHu(R,E,>);
IF E = {} THEN PROOF
ELSE
Choose l = r ∈ E, E := E \ {l = r}
l_1 := rewrite(R,l);
r_1 := rewrite(R,r);
CASE
l_1 ≡ r_1 : HuHu(R,E,>)
l_1 ≡ f(t_1,...,t_n) ∧ f ∈ Σ_C :
   CASE
      r_1 ≡ f(t_1,...,t_n) : HuHu(R,E ∪ {r_i = t_i | i ≤ n},>);
      r_1 ≡ g(t_1,...,t_m) : DISPROOF
   END_CASE
   r_1 ∈ V_S : DISPROOF
END_CASE
l_1 ∈ V_S ∧ r_1 ≡ f(t_1,...,t_n) ∧ f ∈ Σ_C : DISPROOF
l_1 >- r_1 : l_2 := l_1; r_2 := r_1;
   r_1 >- l_1 : l_2 := r_1; r_2 := l_1;
OTHERWISE FAILURE; (* nicht-orientierbare Gleichung *)
(R',E') := SIMPLIFY(R,E,l_2 = r_2);
C := CRITICAL(R',E,l_2 = r_2);
HuHu(R'∪{l_2 = r_2},E'∪C,>)
ENDIF
END PROCEDURE HuHu

Fig. 3.1. Inductionless Induction according to Huet and Hullot

E_2,> terminates with DISPROOF, then at least one equation of E_2 does not hold under E_1. Moreover, if an equation of E_2 does not hold under E_1, then HuHu({},E_1 ∪ E_2,>) terminates with DISPROOF or will fail.

The above algorithm is specialized to abstract data types that are given by constructors as it is the case in the logic of C@S. There are lots of refinements of the algorithm, some of them also implemented in the RRL system, that is interfaced with C@S.

4. Modelling Hardware

In this section, we describe the simplest version of the input language of C@S that is called SHDL. SHDL is an acronym for 'synchronous hardware description language' as it focuses on the description of synchronous systems. SHDL allows to describe systems in a modular way and it offers also the possibility to describe generic structures. In contrast to the logic as given in definition 3.3, abstract data types are defined in SHDL globally, i.e. SHDL has a construct abstype α with c_1,...,c_n that introduces a new data type α that is globally visible. Similarly, SHDL allows to define new functions on these
defined abstract data types according to the primitive recursion theorem that is given by the definition of the data types according to definition 3.5.

4.1 Describing Nongeneric Structures

There are two kinds of (nongeneric) modules in SHDL: one for describing the behavior of base modules and another one for describing the wiring of composed modules.

```
BASIC Module_Name(i1,...,in)
  init(q1) := w1;
  next(q1) := F1((i), (q));
  ...
  init(qs) := ws;
  next(qs) := Fs((i), (q));
OUTPUT
  o1 := F1((i), (q));
  ...
  o1 := Fs((i), (q));
END;
```

```
BASIC D_FlipFlop(inp)
  init(q) := 0;
  next(q) := inp;
OUTPUT
  o1 := q;
  o2 := ~q;
END;
```

```
BASIC And(i1,i2)
OUTPUT
  o := i1 AND i2;
END;
```

Fig. 4.1. Structure of basic modules (left) and two examples (right)

The overall structure of a base module is given in figure 4.1. The lefthand side shows the overall structure of a basic module and the righthand side the description of a D-Flipflop and an AND gate as an example. The module has a name Module_Name and the inputs i1,...,in. A base module may have internal states, which are encoded by boolean state variables q1,...,qs. This means that each tuple of \(\{1, 0\}^s\) is potentially an internal state of the above base module. However it may be the case that some of the potential states are not reachable. The reachability is given by the initial state that is in turn given by the initialization equations \(\text{init}(q_k) := \omega_k\), where \(\omega_k \in \{1, 0\}\). Hence, the initial state of the module is \((\omega_1, \ldots, \omega_s)\). The transition relation is given by the equations \(\text{next}(q_k) := \Omega_k((\vec{i}), (\vec{q}))\) where \(\Omega_k((\vec{i}), (\vec{q}))\) is an arbitrary propositional formula over the variables \(i_1, \ldots, i_n\) and \(q_1, \ldots, q_s\).

Clearly, the semantics of a base module as given in figure 4.1 is the following simple prefix formula, that has as acceptance condition only a single safety property:

\[
\mathcal{P}(\vec{i}) := \left( \exists q_1 \ldots q_s. \left\langle \bigwedge_{k=0}^s \left[ (q_k = \omega_k) \land G \left( \bigwedge_{k=0}^{t} o_k = \Phi_k(i, \vec{q}) \right) \right] \right\rangle \right) \wedge 
\]

\[
\bigwedge_{k=0}^{t} o_k = \Phi_k(i, \vec{q})
\]
We can interpret the above prefix formula either as a rewrite system or as an \( \omega \)-automaton. Hence, SHDL descriptions for nongeneric structures are independent on the proof procedures to verify them. However, it has to be mentioned, that model checking procedures use module descriptions to compute a Kripke structure and hence, SHDL modules are never transformed into prefix formulas. The above prefix formula is just given for defining the semantics of a basic SHDL module.

Note also that the transition relation for basic modules is always deterministic: it is required that for each state variable a transition equation, i.e. an assignment \( \text{next}(q_i) := \varphi_i \) exists. It is allowed that a basic module has no inputs and it is also allowed that a basic module has no internal state variables. In particular, the description of boolean basic gates such as AND-gates do not require internal states. Furthermore, initialisation equations are optional, such that there may be more than one initial state. On the other hand, it is required that each base module has at least one output, otherwise the module does not make sense at all.

\[
\text{MODULE } M(\text{inp}_1, \ldots, \text{inp}_n) \\
C_1 := M_1(v_{1,1}, \ldots, v_{1,n_1}); \\
\vdots \\
C_s := M_s(v_{s,1}, \ldots, v_{s,n_s}); \\
\text{OUTPUT} \\
o_1 := \psi_1; \\
\vdots \\
o_r := \psi_r; \\
\text{SPEC} \\
Spec\_Name_1 \varphi_1; \\
\vdots \\
Spec\_Name_r \varphi_r; \\
\text{END};
\]

\[
\text{MODULE } \text{Mux}(\text{sel}, \text{inp}_1, \text{inp}_2) \\
C_1 : \text{And}(\text{inp}_1, \text{inp}_2); \\
C_2 : \text{Not}(\text{sel}); \\
C_3 : \text{And}(C_2.o, \text{inp}_2); \\
C_4 : \text{Or}(C_1.o, C_3.o); \\
\text{OUTPUT} \\
o := C_4.o; \\
\text{SPEC} \\
S_1 G[o = (\text{sel} \Rightarrow \text{inp}_1 | \text{inp}_2)]; \\
\text{END};
\]

Fig. 4.2. Structure of composed modules (left) and an example (right)

The structure of a composed module and an example of a composed module is given in figure 4.2. SHDL distinguishes between the definition of a module and the use of a module, such as imperative languages distinguish between the definition of a function with formal parameters and the call of the function with some actual parameters. This can also be understood as an object-oriented paradigm: the definition of a module defines a class and uses of the modules define some objects of that class. In the following, the notions of class and object are used, instead of definition and use. The access of an output \( o_i \) of an object \( C_j \) is done by the 'access operator (.)' and is written as \( C_j.o_i \).
The definition of a composed module starts with a module name \textit{Module\_Name} and a formal parameter list \textit{inp}_1, \ldots, \textit{inp}_n. Then a list of objects \(C_1, \ldots, C_s\) follows that belong to the classes \textit{Module\_Name}_1, \ldots, \textit{Module\_Name}_s. It is assumed that these classes are already defined with argument lists of lengths \(n_1, \ldots, n_s\), respectively. In the definition of the objects \(C_1, \ldots, C_s\), the module classes are instantiated with actual parameters \(v_{i,j}\) that can be either an input variable \(\textit{inp}_1, \ldots, \textit{inp}_n\) or an access expression \(C_j.\text{oi}\). In the latter case it is required that (i) the module class \textit{Module\_Name}_j has an output \(\text{oi}\) and that (ii) no cycles occur. This means that if in the actual parameter list of the definition of \(C_j\) an output of \(C_j\) is accessed, then it is required that at least one delay element (with state variables) must be on this cycle.

The definition of the used objects \(C_j\) for the implementation of a composed module implicitly gives the wiring of these modules by the actual parameters that are access expression. In contrast to other hardware description languages, this is done object-oriented, that is by considering the underlying systems instead of the wires between them.

Analogously to basic modules, the outputs of a composed module are given by an equation \(\text{oi} := \psi_i\). However, \(\psi_i\) is now restricted to be either an input variable \(\text{inp}_1, \ldots, \text{inp}_n\) or an access expression \(C_j.\text{oi}\) where \(j \in \{1, \ldots, s\}\). In the latter case it is required that the definition of the module class \textit{Module\_Name}_j has an output \(\text{oi}\). Note that the right hand sides \(\psi_i\) of output equations in basic modules can be arbitrary propositional formulas.

In contrast to basic modules, composed modules can have some specifications. Each specification has a name to identify it and a formula of the specification language. In figure 4.2, the specification of the multiplexer is given in temporal logic, and means that for each computation it will always hold that if \(\text{sel}\) holds then output \(o\) equals to \(\text{inp}_1\), otherwise to \(\text{inp}_2\).

The semantics of a composed module is given by the product automaton of the used objects. Using access expressions, the module hierarchy can be flattened such that the composed module becomes a basic module. This is the same as computing the product automaton of the \(\omega\)-automata of the used basic modules.

### 4.2 Describing Regular Structures

Basic modules and composed modules as described in the last section, allow to describe arbitrary systems. However, regular structures can often be described more compact, if the underlying implementation scheme is given instead of a particular fixed implementation. For example, the implementation of a carry-ripple adder follows a simple cascading implementation scheme that is even more intuitive than an implementation of a 32-bit version as a composed module. SHDL has therefore language constructs for describing regular implementation schemes. These are given in this section.
RECURSIVE Module_Name(p, p_1, ..., p_z; inp_1 : α_1, ..., inp_n : α_n)

CASE p = 0:
  \[ B_1 := B_{\text{Class}_1}(v_{1,1}, \ldots, v_{1,n_1}); \]
  ...
  \[ B_s := B_{\text{Class}_s}(v_{s,1}, \ldots, v_{s,n_s}); \]
OUTPUT
  \[ o_1 := \psi_1; \]
  ...
  \[ o_t := \psi_t; \]
CASE p > 0:
  \[ R_1 := R_{\text{Class}_1}(w_{1,1}, \ldots, w_{1,l_1}); \]
  ...
  \[ R_q := R_{\text{Class}_q}(v_{w,1}, \ldots, v_{w,l_q}); \]
OUTPUT
  \[ o_1 := \Psi_1; \]
  ...
  \[ o_t := \Psi_t; \]
SPEC
  \[ \text{Spec}_\text{Name}_1 \varphi_1; \]
  ...
  \[ \text{Spec}_\text{Name}_r \varphi_r; \]
END;

Fig. 4.3. Structure of recursive module definitions

Figure 4.3 shows the structure of a recursive module definition with the recursion parameter \( p \in \mathbb{N} \). Beneath the inputs \( inp_1, \ldots, inp_n \), an additional parameter list \( p, p_1, \ldots, p_z \) is added to the formal argument list. The first parameter of this list, i.e. \( p \), is the one on which the recursive definition is done. Additionally, it may be the case that there are also other parameters \( p_1, \ldots, p_z \) that are used by components which are required for the definition of the considered module. The list of parameters and the list of inputs are separated by a semicolon. The inputs can be constrained by terms \( \alpha_1, \ldots, \alpha_n \) in order to specify that \( inp_i \) is a list of length \( \alpha_i \). If no constraint is given, the input is supposed to be a boolean valued signal. The constraints \( \alpha_1, \ldots, \alpha_n \) usually depend on the recursion parameter \( p \).

A recursive module defines for each \( p \in \mathbb{N} \) a module that can be described for a fixed \( p \in \mathbb{N} \) by the constructs of the last section, if not other parameters have to be instantiated. The definition of a recursive module is split up in the definition of a base case \( p = 0 \) and the definition of a recursion case \( p > 0 \). Each of these cases is in principle the definition of a composed module. However, in the recursion case it is allowed to use the module class also for instantiation with a parameter smaller than \( p \), usually with \( p - 1 \). It is
mandatory that the names, the types and the number of outputs are the same in each of the two cases.

Recursive modules can be 'unroled' for a fixed parameter \( p \in \mathbb{N} \), i.e. the recursion case of the implementation description is eliminated by successive calls and interpretations of the definition for the fixed parameter. Each instantiation of a parameter of a recursive module with a fixed value \( p \in \mathbb{N} \) is again a module class that can be used for generating objects of this class. If a module contains a variable in the parameter list, it is called a parameterized module.

Recursive modules define also an induction scheme, which is important for the verification of all modules that can be obtained by instantiating the recursion parameter. In particular, one has to show the correctness for the base case \( p = 0 \) and, by using the induction hypothesis, that the module is correct for \( x < p \). It is then necessary to prove that the module is correct for the parameter \( p \). This involves also the elimination of hierarchy, in so far as the objects belonging to the currently considered class and that are used in the definition of the recursion case have to be replaced by one or more of the specifications of the module.

---

**RECURSIVE** Carry_Ripple\( (n; \text{inpl} : n + 1, \text{inp2} : n + 1, \text{carry}) \)

**CASE** \( n = 0 \):

\[ C_1 : \text{FullAdd} (\text{HD} (\text{inpl}), \text{HD} (\text{inp2}), \text{carry}); \]

**OUTPUT**

\[ \text{sum} := [C_1.\text{sum}]; \]
\[ \text{cout} := C_1.\text{cout}; \]

**CASE** \( n > 0 \):

\[ C_1 : \text{FullAdd} (\text{HD} (\text{inpl}), \text{HD} (\text{inp2}), C_2.\text{cout}); \]
\[ C_2 : \text{Carry_Ripple} (n - 1; \text{TL} (\text{inpl}), \text{TL} (\text{inp2}), \text{carry}); \]

**OUTPUT**

\[ \text{sum} := (C_1.\text{sum} \triangleright C_2.\text{sum}); \]
\[ \text{cout} := C_1.\text{cout}; \]

**SPEC**

\[ S_1 \; \text{AG}[\neg \text{carry} \rightarrow ((\text{cout} \triangleright \text{sum}) \sim_{\mathbb{N}} (\text{inpl} \triangleright \text{inp2} ))]; \]
\[ S_2 \; \text{AG}[\text{carry} \rightarrow ((\text{cout} \triangleright \text{sum}) \sim_{\mathbb{N}} \text{INC} ((\text{inpl} \triangleright \text{inp2} )))]; \]

**END;**

---

*Fig. 4.4. Example of a recursive module*

Figure 4.4 gives the recursive definition of a carry-ripple adder in SHDL. The module is parameterized with a single parameter \( n \) that is the number of bits of the inputs that are to be added. Hence, the inputs \( \text{inpl} \) and \( \text{inp2} \) are lists of bits of the length \( n + 1 \) and the third input \( \text{carry} \) is a simple boolean valued signal. The outputs are a list of length \( n \) that has the name \( \text{sum} \) and a boolean valued output \( \text{cout} \). The latter is high exactly when an overflow
occurred (it may also be viewed as the most significant bit of the sum) and \textit{sum} contains the first \( n \) bits of the sum of the inputs.

In the base case \((n = 0)\), the system consists simply of a full adder, whose inputs are \( \text{inpl}[0] \) and \( \text{inp2}[0] \) (note that in this case \( \text{inpl} = [\text{inpl}[0]] \) and \( \text{inp2} = [\text{inp2}[0]] \) holds). Note that the output \( C_1.\text{sum} \) of the full adder is a boolean valued signal that has to be put in a list such that the types of the outputs are correct. The recursion case the inputs \( \text{inpl} = [\text{inpl}[0], \ldots, \text{inpl}[p-1]] \) and \( \text{inp2} = [\text{inp2}[0], \ldots, \text{inp2}[p-1]] \) are split up. The lower \( p - 1 \) bits \( \text{TL}(\text{inpl}) = [\text{inpl}[1], \ldots, \text{inpl}[p-1]] \) and \( \text{TL}(\text{inp2}) = [\text{inp2}[1], \ldots, \text{inp2}[p-1]] \) are fed into a carry-ripple adder \( C_2 \) of size \( p - 1 \) and the most significant bits \( \text{HD}(\text{inpl}) = \text{inpl}[0] \) and \( \text{HD}(\text{inp2}) = \text{inp2}[0] \) are fed into a full adder \( C_1 \). The overflow output of \( C_1 \) is wired to the carry-in input of \( C_2 \). The overflow output is then the overflow output of \( C_1 \) and the sum is the sum computed by \( C_2 \) with the extension of the sum of \( C_1 \) at the left hand side of \( C_2.\text{sum} \).

Additionally, some specifications are given for the system that are not considered in detail here. \( S_1 \) states that the systems output \textit{sum} and \textit{cout} can be viewed as the sum of the inputs when \textit{carry} is low, otherwise it can be viewed as the increment of the sum of the inputs.

It is important to assure the consistency of the definition of a recursive module. This involves a type checking together with the constraints on list lengths given by the recursion parameter \( p \). For example, in figure 4.4 the inputs of the carry-ripple adder \( \text{inpl} \) and \( \text{inp2} \) are declared in the header as lists of length \( p \) (length constraint). The given component lists make then clear that these lists are lists of boolean values (type constraint). In the component declaration of the module object \( C_2 \) in the recursion case, it is necessary that the conditions \( ||\text{TL}(\text{inpl})|| = ||\text{inpl}|| - 1 \) and \( ||\text{TL}(\text{inp2})|| = ||\text{inp2}|| - 1 \) are fulfilled. Otherwise, the definition of the module can run into the risk of being inconsistent and hence of making no sense at all. These consistency conditions can be generated automatically for each definition of a recursive module, but need to be proved by hand (usually it requires interactive proofs, but can be done with a high degree of automation).

Figures 4.5 and 4.6 show the definition of an Omega-network as a more complicated example of a generic structure. Figure 4.5 defines the perfect shuffle function and a switching circuit. The latter one has two inputs \( \text{inpl}, \text{inp2} \) and an additional control input \textit{sel}. The outputs \( \alpha_1, \alpha_2 \) directly correspond to \( \text{inpl}, \text{inp2} \) if \textit{sel} is high, or to \( \text{inp2}, \text{inpl} \), otherwise. \textit{SwitchN}(\( n; s, a \)) is a column of \( n + 1 \) switches, hence there are \( n + 1 \) control inputs \( s \) for mapping the \( 2(n + 1) \) inputs to the \( 2(n + 1) \) outputs. The modules given in figure 4.6 combine these modules to the Omega-network. \textit{NetHelp}(\( m, n; s, a \)) has two parameters \( m \) and \( n \) and as inputs a list \( s \) that contains \( m + 1 \) boolean lists and a boolean list \( a \) of length \( 2(n + 1) \). The type of \( s \) is firstly determined by the length restriction \( s : m + 1 \) in the parameter definition of the module \textit{NetHelp}. The module instantiation \( S : \text{SwitchN}(n; \text{HD}(s), a) \) requires that \( \text{HD}(s) \) is a boolean list of length \( n + 1 \).
RECURSIVE Perfect_Shuffle(n; a : n + 1, b : n + 1)
CASE n = 0:
   OUTPUT
   out := [EL0(a), EL0(b)];
CASE n > 0:
   P := Perfect_Shuffle(n - 1; TL(a) TL(b));
   OUTPUT
   out := (EL0(a) P out);
END;

MODULE Switch(s, inp1, inp2)
   C1 := Mux(sel, inp1, inp2);
   C2 := Not(sel);
   C3 := Mux(C2.0, inp1, inp2);
   OUTPUT
   o1 := C1.o;
   o2 := C2.o;
END;

RECURSIVE Switch_N(n; s, a : 2(n + 1))
CASE n = 0:
   C1 := Switch(HD(s), EL0(a), EL1(a));
   OUTPUT
   out := [C1.o1, C1.o2];
CASE n > 0:
   C1 := Switch(HD(s), EL0(a), EL1(a));
   C2 := Switch_N(n - 1; TL(s), TL(TL(a)));
   OUTPUT
   out := (C1.o1 C2.out);
END;

Fig. 4.5. Perfect shuffle function and a column of switches as recursive modules

Net_Help(m, n; s, a) simply consists of m + 1 columns of Switch_N and Perfect_Shuffle modules. This is in principle the implementation of the Omega-network. However, as Omega-networks are defined to consist of m + 1 columns of perfect shuffles with switching elements of length 2^{m+1}, we have to constrain the parameter n to 2^{m+1} - 1. This is the task of the module Omega_Net. Hence, the argument list a in the module Omega_Net has now the length 2(2^{m+1} - 1 + 1) = 2^{m+2}. Note that the definition of the module Omega_Net is not done recursively. A correctness proof of such generic modules requires usually to prove some lemmata about its components that are combined by some interactive rules to obtain the final correctness result.
5. Specification and Proof

5.1 The Unifying Principle

In general, a specification in C@S may consist of a combination of temporal operators and abstract data type expressions as presented in section 3.1. Clearly, it is useful to separate the temporal and the data part such that they can be tackled with different approaches. For example, the specification 
\[(C = A + 1) W (B = 0)\] (\(A, B, C\) are natural numbers and + is a defined function symbol for the usual addition) is transformed into \([\varphi_1 W \varphi_2]\) where \(\varphi_1\) and \(\varphi_2\) are new variables of type \(\mathbb{B}\) with the meaning \(\varphi_1 := (C = A + 1)\) and \(\varphi_2 := (B = 0)\). These new variables are added by let-expressions similar to those of functional programming languages as follows:

\[
\text{let } \left[ \begin{array}{c} \varphi_1 \\ \varphi_2 \\ \end{array} \right] := \left[ \begin{array}{c} C = A + 1 \\ B = 0 \\ \end{array} \right] \text{ in } [\varphi_1 W \varphi_2] \text{ end}
\]

In general, all specifications in C@S follow the template given below:

\[
\text{C.SPEC}(\overline{i}, \overline{o}) := \begin{pmatrix}
\text{let} \\
\left[ \begin{array}{c} \varphi_1 \\ \vdots \\ \varphi_n \\ \end{array} \right] := \left[ \begin{array}{c} \Theta_1(\overline{i}, \overline{o}) \\ \vdots \\ \Theta_n(\overline{i}, \overline{o}) \\ \end{array} \right] \\
\text{in} \\
\Phi(\varphi_1, \ldots, \varphi_n) \\
\text{end}
\end{pmatrix}
\]
274 Klaus Schneider and Thomas Kropf

The above specification consists of a temporal abstraction \( \Phi \) and the data equations \( \Theta_1(t, o), \ldots, \Theta_n(t, o) \). \( \Phi \) is a propositional linear temporal logic formula containing only the propositional variables \( \varphi_1, \ldots, \varphi_n \), i.e. no data expressions may occur. The above separation between temporal and data abstraction parts follows the usual distinction of digital systems into a controller and a data path. In some way, \( \Phi(\varphi_1, \ldots, \varphi_n) \) corresponds to the specification of the control flow, while the data equations correspond to the specification of the data flow. Of course, it is usually not the case that these specification can be verified independently from each other.

As it has been shown in section 3., there exist proof procedures for temporal propositional logic and for proving the consistency of rewrite systems with abstract data types. However, it is difficult to combine both classes of proof procedures. The next subsection explains some strategies how this is done in the C@S system.

5.2 Strategies for Verifying Temporal Behavior

In this section, the \( W \) operator is used to illustrate different translation procedures of C@S. All presented procedures are able to capture full linear temporal logic as defined in definition 3.3. The main strategy is \( \Phi_{PF} \) for translating verification goals into prefix formulas. If the corresponding prefix formula consists only of a safety property, then a simple combination with proof procedures for term rewrite systems can be achieved by interpreting the prefix formula as a term rewrite system.

If, on the other hand, the corresponding prefix formula has also liveness or fairness properties in its acceptance condition, then the strategy \( \Phi_{IV} \) can be used to eliminate some temporal operators. \( \Phi_{IV} \) can also be used to reduce the complexity of the result of \( \Phi_{PF} \) by providing more proof information by additional invariants. C@S has also other proof procedures for temporal reasoning as \( \Phi_N \), that translates temporal logic to arithmetic and \( \Phi_{\mu} \) that translates temporal logic to fixpoint equations. These proof procedures are however not given in this article.

5.2.1 Strategy \( \Phi_{PF} \). It is straightforward to define various temporal operators by prefix formulas. For example, the \( W \)-operator can be defined as follows:

\[
\neg_{def} [x W b] := \left( \exists q. (q = 0) \land G (Xq = q \lor b) \land \frac{G \neg b \lor x \lor q}{G \neg b \lor x \lor q} \right)
\]

The strategy \( \Phi_{PF} \) uses these definitions to translate the temporal abstraction \( \Phi \) into an equivalent prefix formula \( \mathcal{H}_{\Phi} \). In order to capture full LTL, additional universal quantification over input variables and additional fairness constraints are required for prefix formulas [Schn96b]. Section 5.3 explains in detail how strategy \( \Phi_{PF} \) works.
5.2.2 Strategy $\Phi_{IV}$. Provided that invariants are given for some occurrences of temporal operators, this strategy eliminates the corresponding temporal operators by invariant rules of $\text{C@S}$. For example, the invariant rule of the $W$ operator is as follows [ScKK94a]:

$$[x \ W b] := \left( \begin{array}{c} \exists J. \\
J \land \\
g \left( -b \land J \rightarrow x J \right) \land \\
g \left( b \land J \rightarrow x \right) \end{array} \right)$$

The bound variable $J$ is called an invariant and the subformulas have the following meaning: The first one states that the invariant $J$ holds at the beginning of the computation, the second one states that if the termination condition $b$ does not hold, the invariant $J$ implies that the invariant $J$ holds also at the next point of time, and the third subformula states that if the termination condition $b$ and the invariant $J$ hold at a certain time, then the condition $x$ holds also at this point of time.

As already mentioned, invariant rules are applied for the following purposes: first, they are used to cut data dependencies between control and data part of the verification goals. Second, they often eliminate the introduction of fairness constraints by $\Phi_{PF}$, such that an application of $\Phi_{PF}$ yields in prefix formulas with only a safety property that can be checked by traditional rewrite methods. A third purpose is that even in case that model checking techniques are used, the complexity of these procedures can be significantly reduced as shown in the experimental results section.

5.3 Translating LTL Formulas into Fair Prefix Formulae

In this section, the translation method for LTL as defined in section 3.3 into universally quantified fair prefix formulas is presented. The translation of a LTL formula $\varphi$ involves the following steps:

1. Computation of the negation normal form
2. Computation of the prenex next normal form $\varphi_p$ with kernel $\psi_p$
3. Reduction of the kernel $\psi_p$ to quantified $W$ normal form
4. Computation of the $X$- and $W$-closures and generation of fairness constraints by a bottom-up traversal through the syntax tree.

The negation normal form has to be computed since negations of SPFs might cause an exponential blow-up. Nevertheless, SPFs are closed under arbitrary boolean operations as stated in the following theorem.

Theorem 5.1 (Boolean Closure of Simple Prefix Formulas). For all SPFs $P_1(\bar{i}), P_2(\bar{i})$ there are SPFs $P_\neg(\bar{i}), P_\land(\bar{i}), P_\lor(\bar{i})$ such that $\models P_\neg(\bar{i}) = \neg P_1(\bar{i}), \models P_1(\bar{i}) \land P_2(\bar{i}) = P_\land(\bar{i})$ and $\models P_1(\bar{i}) \lor P_2(\bar{i}) = P_\lor(\bar{i})$ holds.
Proof. According to the closure lemma [Schn96a], it is sufficient to show that all boolean operations on SPF acceptance conditions can be expressed as SPF. The proof of the closure under conjunction is trivial. To show the closure under disjunction, an equivalent SPF for the following formula has to be found:

\[
\left( \bigwedge_{j=0}^{a_1} [G \phi_{1,j}] \lor [F \psi_{1,j}] \right) \lor \left( \bigwedge_{k=0}^{a_2} [G \phi_{2,k}] \lor [F \psi_{2,k}] \right)
\]

Using distributivity laws for \( \lor \) and \( \land \), the following equivalent formula is obtained:

\[
\bigwedge_{j=0}^{a_1} \bigwedge_{k=0}^{a_2} ([G \phi_{1,j}] \lor [G \phi_{2,k}] \lor [F \psi_{1,j}] \lor [F \psi_{2,k}])
\]

Disjunctions of liveness properties can be reduced to a single liveness property using the law \([F \psi_{1,j}] \lor [F \psi_{2,k}] = F (\psi_{1,j} \lor \psi_{2,k})\). Reducing the disjunction of safety properties is not possible in this manner. This can be done by introducing new state variables \(p_j, q_k\) to watch the corresponding safety properties \(\phi_{1,j}\) and \(\phi_{2,k}\), respectively. These 'watchdog variables' are initially set to 0 and switch to 1 if the corresponding safety property is 0 at a certain point of time. After switching to 1 the watchdog variables stay on the value 1 regardless what happens. Using the watchdog variables, \([G \phi_{1,j}] \lor [G \phi_{2,k}]\) is equivalent to \(G(\neg p_j \lor \neg q_k)\). Thus the following SPF is obtained:

\[
\exists p_0 \ldots p_{a_1}, q_0 \ldots q_{a_2} \cdot \\
\bigwedge_{j=0}^{a_1} \left[ (p_j = 0) \land G (X p_j = p_j \lor \neg \phi_{1,j}) \right] \land \\
\bigwedge_{k=0}^{a_2} \left[ (q_k = 0) \land G (X q_k = q_k \lor \neg \phi_{2,k}) \right] \land \\
\bigwedge_{j=0}^{a_1} \bigwedge_{k=0}^{a_2} \left( [G(\neg p_j \lor \neg q_k)] \lor [F(\psi_{1,j} \lor \psi_{2,k})] \right)
\]

To show the closure under negation, the closure lemma is first used to shift the negation over the transition relation. Then negation is shifted inside the safety and liveness properties, such that the acceptance condition \(\bigvee_{j=0}^{a} [F \neg \Phi_j(\eta)] \land [G \neg \Psi_j(\eta)]\) is obtained. As each safety and each liveness property is itself a SPF (without state transitions), this is just the disjunction of \(a + 1\) SPF. As already proved, this can be expressed as an equivalent SPF. As each boolean operation can be expressed by \(\land, \lor\) and \(\neg\) the boolean closure of SPF can be concluded in general.

Conjunctions of SPF can be performed in constant time, while performing disjunctions requires at most quadratic space and time. Negations, on the other hand, can blow up a SPF by an exponential factor. Thus, negations of SPF are avoided in the translation procedure by converting the given LTL formula first into negation normal form as follows:
Theorem 5.2 (Negation Normal Form (NNF)). The application of the following theorems converts each LTL formula into an equivalent one where negation symbols only occur in front of variables:

- \( \neg (\neg \varphi) = \varphi \)
- \( \neg (\varphi \land \psi) = \neg \varphi \lor \neg \psi \)
- \( \neg (\varphi \lor \psi) = \neg \varphi \land \neg \psi \)
- \( \neg X \varphi = X \neg \varphi \)
- \( \neg G \varphi = F \neg \varphi \)
- \( \neg F \varphi = G \neg \varphi \)
- \( \neg [x W b] = [\neg x] W b \land [F b] \)
- \( \neg [x U b] = [\neg b] W (x \rightarrow b) \land [F(x \rightarrow b)] \)

The definition and existence of the prenex next normal form is based on the following theorem. The computation itself is straightforward and therefore not presented in detail.

Theorem 5.3 (Prenex Next Normal Form (PNNF)). For every LTL formula \( \Phi \), there exists an equivalent formula of the following form, where \( \Phi \) is a LTL formula without \( X \)-operators:

\[
\exists q_1 \ldots q_n. \left( \bigwedge_{k=1}^{n} G (Xq_k = x_k) \right) \land [X \ldots X\Psi]
\]

The variables \( q_k \) must not occur in \( \Phi \) and the variables \( x_k \) are either variables occurring in \( \Phi \) or one of the variables \( q_k \). The formula \( \Psi \) is called the kernel of the above PNNF.

The \( X \)-operator commutes with all other operators, i.e. \( (Xx) \land (Xy) = X(x \land y) \), \( [(Xy) W (Xx)] = X[y W x] \), etc. However, if only one argument of a binary operator has a leading \( X \)-operator, as e.g. in \( [(Xy) W x] \), these laws cannot be used directly. In this case, a new variable \( q \) is introduced by defining \( G(Xq = x) \), such that the formula \( \exists q. G(Xq = x) \land [(Xy) W (Xq)] \) is obtained. After that the \( X \)-operators can be shifted outwards. For example, the PNNF of \( X[b W x] \land c \) is

\[
\exists q_1 q_2 q_3. \quad G(Xq_1 = b) \land G(Xq_2 = c) \land G(Xq_3 = q_2) \land XX (q_1 W a) \land q_3
\]

As the initial values of the new variables \( q_k \) in the above theorem are not considered for the evaluation of the truth value of \( \Psi \), they may be arbitrarily set to 1 or 0. The equations \( G(Xq_k = x_k) \) can therefore be viewed as transition equations of a transition relation and hence, it is sufficient to translate the remaining formula \( [X \ldots X\Psi] \) into a SPF. The following closure theorem for the \( X \)-operator shows how this can be done, if the \( \Psi \) can be translated into a FPF:

Theorem 5.4 (X-Closure of Fair Prefix Formulas). Given the FPF \( \mathcal{P}(\vec{z}) \) of definition 3.6, the following FPF is equivalent to \( XP(\vec{z}) \):
A proof can be found in [Schn96a]. Thus, it remains to show how LTL formulas without X-operators can be translated into FPF. This is based on the following closure theorem for the temporal operator W, which can also be adapted to W.

**Theorem 5.5 (W-Closure of Fair Prefix Formulas).** Given the \( \mathcal{P}(\bar{t}) \) of definition 3.6 and a propositional formula \( b \), the following FPF is equivalent to \( [\mathcal{P}(\bar{t}) \ W b] \):

\[
\exists p_1 \ldots p_s. \left( [(p = 0) \land G (Xp = 1)] \land \bigwedge_{k=0}^{s} \left[ (q_k = \omega_k) \land G (Xq_k = \left( p \Rightarrow \Omega_k(\bar{t}, \bar{q}) \mid \omega_k) \right) \right] \land \left( \bigwedge_{l=0}^{f} GF\xi_l(\bar{t}, \bar{q}) \rightarrow \bigwedge_{m=0}^{a} \left[ G(p \rightarrow \Phi_m(\bar{t}, \bar{q})) \right] \lor \left[ F(p \land \Psi_m(\bar{t}, \bar{q})) \right] \right) \right)
\]

Again, a proof of the theorem can be found in [Schn96a]. However, these theorems do not allow to derive closure theorems for the remaining temporal operators from the above theorem, since the assumption that the event \( b \) has to be propositional is not fulfilled in general. In order to handle the closure for these operators, the following theorem is used to reduce these operators into equivalent quantified W expressions with a propositional event. The key of this theorem is the introduction of new signals as events, which may become true in a special interval.

**Theorem 5.6 (Reduction to Quantified W expressions).** Given that \( b \) is propositional, the following equations hold (\( a \) is a new variable):

1. \( [G\varphi] = \forall a. [\varphi \ W a] \)
2. \( [F\varphi] = \exists a. [\varphi \ W a] = \exists a. Fa \land [\varphi \ W a] \)
3. \( [\varphi U b] = \forall a. [(-a) \ U b] \lor [\varphi \ W a] \)
4. \( [[\forall a. \mathcal{P}(a)] \ W b] = \forall a. [[\mathcal{P}(a)] \ W b] \)
5. \( [[\exists a. \mathcal{P}(a)] \ W b] = \exists a. [[\mathcal{P}(a)] \ W b] \)

The first three items of the above theorem eliminate the temporal operators G, F and U if the event of U is propositional. Finally, the last two equations allow to shift quantifiers outside W-operators such that a prenex normal form...
The COS System

\begin{verbatim}
VAL V := \{};

FUNCTION add_var(\Theta) = {a := new_var; V := V \cup \{(a, \Theta)\}; return a; }

FUNCTION QW(\varphi) =
  \begin{enumerate}
  \item \prop(\varphi) : return \exists p. ((q = 0) \land G (Xq = 1)) \land G (q \lor p)
  \item \varphi_1 \land \varphi_2 : return SPF_CONJ(QW(\varphi_1),QW(\varphi_2))
  \item \varphi_1 \lor \varphi_2 : return SPF_DISJ(QW(\varphi_1),QW(\varphi_2))
  \item X\varphi_1 : return SPF_NEXT(QW(\varphi_1))
  \item G\varphi_1 : if prop(\varphi_1) then return G\varphi_1
      else \{ a := add_var(V); return \[QW(\varphi_1) W a]\}
  \item F\varphi_1 : if prop(\varphi_1) then return F\varphi_1
      else \{ a := add_var(V); return Fa \land [QW(\varphi_1) W a]\}
  \item [\varphi_1 U \varphi] : if prop(\varphi_1) then return 3q. \[([q = 0] \land G (Xq = q \lor \varphi)) \land G (q \lor \varphi_1)\]
      else \{ a := add_var(V); return QW(\[\neg \varphi] U \varphi) \land [QW(\varphi_1) W a]\}
  \item [\varphi_1 W \varphi] : if prop(\varphi_1) then return 3q. \[([q = F] \land G (Xq = q \lor \varphi)) \land G (q \lor \neg \varphi \lor \varphi_1)\]
      else return SPF_WHEN(b,QW(\varphi_1))
  \end{enumerate}

FUNCTION LTL2QWHEN(\varphi) =
  \{ \forall := \{}; \exists := QW(\varphi); return mk_quantify(\forall,\exists); \}
\end{verbatim}

**Fig. 5.1.** Algorithm for translating a subset of LTL to quantified SPF

...can be obtained similar to the construction of the prenex normal form for first order logic. A proof of the theorem can be found in [Schn96a]. This directly leads to the following theorem:

**Theorem 5.7.** Given a LTL formula \( \Phi(\vec{i}) \) in negation normal form such that for all subformulas \([x \lor b] \) and \([x U b] \) the event \( b \) is propositional, then there is a SPF \( P(\vec{i}, \vec{a}) \) with new variables \( \vec{a} \) such that for some \( \Theta_j \in \{\forall,\exists\} \) the equation \( \Phi(\vec{i}) = \Theta_1 a_1 \ldots \Theta_n a_{n||\vec{a}}.P(\vec{i}, \vec{a}) \) holds.

**Proof.** \( \Theta_1 a_1 \ldots \Theta_n a_{n||\vec{a}}.P(\vec{i}, \vec{a}) \) is computed by the function LTL2QWHEN of figure 5.1. The proof is done by structural induction and follows directly the implementation of LTL2QWHEN. In the induction steps, the closure theorems for the boolean operators, X, and W are used and the induction basis follows from the following theorems, where \( x \) and \( b \) have to be propositional:

- \([x \lor b] := \exists q. ([q = 0] \land G (Xq = q \lor b)) \land G (q \lor \neg b \lor x)\)
- \([x U b] := \exists q. ([q = 0] \land G (Xq = q \lor b)) \land G (q \lor b \lor x)\)
- \(x = \exists p. ([p = 0] \land G (Xp = 1)) \land G (x \lor p)\)

Hence, the closure of SPFs under boolean operators, X and W allows together with theorem 5.6 to construct for certain LTL formulas an equivalent quan-
tified SPF. In the following, it is shown how arbitrary LTL formulas can be translated to universally quantified FPFs. Thus the following drawbacks of the function LTL2QWHEN have to be circumvented:

1. LTL2QWHEN can only translate LTL formulas, whose events are propositional, i.e. for all subformulas \([x W b]\) and \([x U b]\) the event \(b\) has to be propositional.
2. LTL2QWHEN does not always produce universally quantified formulas. Unfortunately, there is no ‘simple’ decision procedure for arbitrarily quantified SPFs. Since \(\forall v_1 \ldots v_n. P(\bar{v}, \bar{v})\) holds iff \(\forall P(\bar{v}, \bar{v})\) holds, the validity of universally quantified SPFs can be computed by a decision procedure for SPFs.

Both problems are circumvented by the following trick: whenever one of the above problems is detected, the corresponding subformula \(\varphi\) is replaced by a new variable \(\ell\). Of course, it has to be guaranteed that \(\ell\) behaves always equivalent to \(\varphi\). This is done simply by adding the assumption \(G(\ell = \varphi)\), according to the theorem: \(\Phi(\varphi) = \forall \ell. [G(\ell = \varphi)] \rightarrow \Phi(\ell)\). The detailed computation of the assumptions is given by the algorithm of figure 5.2 and stated in the following theorem:

**Theorem 5.8.** Given an arbitrary LTL formula \(\Phi(\bar{v})\) in negation normal form, the function LTL2FLTL in figure 5.2 generates a set of formulas \(\mathcal{E} = \{G(\ell_1 = \varphi_1(\bar{v}, \bar{\ell})), \ldots, G(\ell_n = \varphi_n(\bar{v}, \bar{\ell}))\}\) with new variables \(\ell_j\) (not occurring in \(\Phi(\bar{v})\)) and a simple prefix formula \(P(\bar{v}, \bar{\ell}, \bar{a})\) such that the following holds:

\[
\Phi(\bar{v}) = \forall \ell_1 \ldots \ell_n. \bigwedge_{j=1}^{n} [G(\ell_j = \varphi_j(\bar{v}, \bar{\ell}))] \rightarrow \forall a_1 \ldots a_{\|\bar{a}\|}. P(\bar{v}, \bar{\ell}, \bar{a})
\]

Moreover, the formulas \(\varphi_j(\bar{v}, \bar{\ell})\) are of one of the following forms: \(Gx\), \(Fx\), \([x U b]\) or \([x W b]\), where both \(b\) and \(x\) are propositional.

**Proof.** Given a LTL formula \(\varphi\) in negation normal form, the function top in figure 5.2 replaces all temporal subformulas of \(\varphi\) by new variables and adds corresponding assumptions to the set \(\mathcal{E}\). For example, top([\([Fy] W [Gx]\)] \land FGz) adds the assumptions \(G(\ell_1 = Gx), G(\ell_2 = Fy), G(\ell_3 = [\ell_2 W \ell_1])\), \(G(\ell_4 = Gz)\), \(G(\ell_5 = FL4)\) to \(\mathcal{E}\) and finally returns \(\ell_3 \land \ell_5\). Note that top(\(\varphi\)) is always propositional and note also that top(\(\varphi\))=\(\varphi\), if \(\varphi\) is propositional. top is used by the function GenFair to replace non-propositional events \(b\) of subformulas \([x U b]\) and \([x W b]\) by propositional ones. Thus the first of the two problems of LTL2QWHEN is circumvented. The introduction of \(\exists\)-quantifiers by LTL2QWHEN arises only when a subformula \(F\varphi_1\) is replaced by its equivalent WHEN expression. The call of the function top in the function GenFair (in case \(\varphi = F\varphi_1\)) eliminates this case. Hence, the result of the function GenFair is – under the assumptions of the set \(\mathcal{E}\) – an equivalent formula which can be translated by LTL2QWHEN to a universally quantified SPF.
It remains now to translate the assumptions $G(\ell_j = \varphi_j)$ into fairness constraints. As $Gx = [0 W (-x)]$, $Fx = [1 W x]$, $[x U b] = [b W (x \rightarrow b)]$ holds, it is sufficient to be able to translate $G (\ell = [x W b])$ and $G (\ell = [x W b])$ to fairness constraints. As $(\ell = [x W b]) = (-\ell = [(\neg x) W b])$ holds, it is even only necessary to be able to translate $G (\ell = [x W b])$ into a fairness constraint. How this is done, is explained in the following lemma.

**Lemma 5.1 (Replacing Definitions by Fairness Constraints).**

$G (\ell = [x W b])$ is equivalent to:

$$
\begin{align*}
(p = 1) \land G (Xp = (\ell \land q \land [x \lor \neg b]) \lor (-\ell \land q \land b \land \neg x)) \land \\
(q = 1) \land G (Xq = (-\ell \land q \land \neg [x \land b]) \lor (\ell \land p \land b \land x)) \land \\
G F p
\end{align*}
$$

The above theorem has been constructed by computing the accepting Büchi automata according to [MaPn87b] and has been formally proved in the HOL system. A proof can be found in [Schn96a].
In order to reduce a LTL formula $\Phi(\vec{i})$ into a universally quantified FPF, simply apply the function LTL2FLTL to obtain $\forall \vec{\ell}. \bigwedge_{j=1}^{n} [G(\ell_j = \varphi_j(\vec{i}, \vec{\ell}))] \rightarrow \forall \vec{a}. \mathcal{P}(\vec{i}, \vec{\ell}, \vec{a})$. After that, shift the quantification $\forall \vec{a}$ outwards to $\forall \vec{\ell}$ and replace the assumptions $G(\ell_j = \varphi_j(\vec{i}, \vec{\ell}))$ according to the above lemma, where the new transitions are added to the transitions of $\mathcal{P}(\vec{i}, \vec{\ell}, \vec{a})$.

5.4 Strategies for Verifying Abstract Data Types

In this section, different strategies for handling the data expressions in verification goals are discussed. In order to process a higher order data type $C$ with a digital system, a finite number of boolean values is usually collected and interpreted as single unit according to an interpretation function $\Phi_C$. For example, if natural numbers are to be processed, then a tuple $[a_n, \ldots, a_0]$ of boolean values is often interpreted as the natural number $\Phi_N([a_n, \ldots, a_0]) := \sum_{i=0}^{n} 2^i \times a_i$. In general, each specification that uses an abstract data type $C$ can be described at three levels of abstractions as shown in figure 5.3:

- The data operations can be formulated directly on the abstract data type $C$ with operations $\pi_C$ that are defined on that data type, as e.g. $\Phi_N([b_2, b_1, b_0]) = \Phi_N([a_1, a_0]) + 1$. This is the most readable version, but as the abstract data type $C$ does not occur in hardware (structure) descriptions, the verification requires to lift bitvector operations $\pi_{BV}$ used in the hardware description in SHDL by the data interpretation function $\Psi_C$ to a corresponding operation $\pi_C$ on the data type $C$.
- Data operations may also be formulated at the 'bitvector level', where a bitvector is defined to be a boolean list with *an arbitrary length*. The set
of bitvectors is denoted by $\mathbb{B}V$ in the following. Using predefined bitvector operators $\pi_{\mathbb{B}V}$, the above example looks like $[b_2, b_1, b_0] = \text{INC}([a_1, a_0])$, where $\text{INC}$ is a bitvector operation that is defined as follows ($(b \triangleright B)$ appends a boolean term $b$ at the lefthand side of the bitvector $B$):

\[
(\text{ALLONE} ([]) = [1]) \land (\text{ALLONE} ((b \triangleright B)) = b \land \text{ALLONE} (B))
\]

\[
(\text{INC} ([]) = [1]) \land (\text{INC} ((b \triangleright B)) = ((\text{ALLONE} (B) \oplus b) \triangleright \text{INC} (B)))
\]

This level has many advantages: first its readability is comparable to descriptions at the abstract level $C$. The only data type that is really used is 'bitvector' $\mathbb{B}V$, hence it is sufficient to have a decision procedure for bitvectors. This circumvents the problem of defining for each data type $C$ an interpretation function $\Psi_C$. For fixed lengths of bitvectors, it is also possible to translate these descriptions to propositional logic such that tautology checking can serve as a decision procedure.

The data operations can directly be formulated as propositional formulas on the bits $a_n, \ldots, a_0$, where $n$ has to be a concrete number in this case. Using two bits, the above example is represented as $(b_0 = \neg a_0) \land (b_1 = a_1 \oplus a_0) \land (b_2 = a_1 \land a_0)$. The advantage of this description is that simple tautology checking can be used as decision procedure for data types. However, these descriptions are not readable at all and do not allow $n$-bit specifications.

We usually prefer the second level for the verification, although the C@S logic would also allow us to use the most abstract level or the propositional level. For this reason, the following strategies can be found in C@S for verifying abstract data types:

**Strategy $\Theta_B$.** This strategy transforms the equations $\Theta_1, \ldots, \Theta_n$ of a specification as given on page 273 into propositional formulas $\Theta_1^{\text{prop}}, \ldots, \Theta_n^{\text{prop}}$ that can be substituted in the temporal abstraction $\Phi$. Of course, this strategy does only work with non-generic systems.

**Strategy $\Theta_{BV}$.** This strategy translates operations on abstract data types such as addition on natural numbers in corresponding operations on bitvectors. If the specification is already given on bitvectors instead of (real) abstract data types, then this strategy is not required.

**Strategy Induct.** This strategy invokes an inductionless induction procedure to prove a specification that is given at the bitvector level. It is assumed that all operations are also given at this level or that strategy $\Theta_{BV}$ has been invoked before.

**Strategy Decompose.** If the system is recursively defined, then this strategy decomposes the verification goal into separate subgoals by applying a suitable induction rule which directly reflects the definition of the system. According to the induction hypothesis, the system structure is replaced by the system’s specification in the induction step. Note that systems which are built up non-recursively but use recursively defined modules are not changed by this strategy.
Strategy Interact. This strategy allows to apply all interactive rules of the COS. Hence, this strategy is not specialized for any system structure or any verification goal and allows arbitrary manipulations of the goal. It is however the intention to invoke this strategy only if a fully automated proof is not possible. In such cases, strategy Interact is used to prove additional lemmas or to change the design hierarchy such that one of the other rules can be applied.

5.5 Combining the Strategies

The above strategies have to be combined specifically for each verification problem. In general, the choice of the suitable strategy can be done according to the algorithm Verify below. \( \text{imp} \) is the implementation description, \( \text{sp} \) is the list of specifications for the currently considered module and \( \text{inv} \text{.list} \) is a possibly empty list of given invariants.

\[
\text{FUNCTION Verify}(\text{imp}, \text{sp}, \text{inv}\text{.list})
\]

\[
\text{IF Recursive\_Definition}(\text{imp})
\]

\[
\text{THEN MAP Verify(Decompose(imp, sp, inv\text{.list}))}
\]

\[
\text{ELSE}
\]

\[
\text{let sp1} = \Phi_{PF}(\Phi_{IV} (\text{inv}\text{.list}, \text{sp}))
\]

\[
\text{in IF fixed\_bv\_length(imp)}
\]

\[
\text{THEN HW\_Decide(HW\_IMP(imp, } \Theta_B (sp1))))
\]

\[
\text{ELSE}
\]

\[
\text{IF one\_safety(sp1)}
\]

\[
\text{THEN Induct(imp, sp1)}
\]

\[
\text{ELSE Interact(imp, sp, inv\text{.list})}
\]

Verify first tries to decompose recursively defined systems by applying induction rules. The induction principle follows directly the structure of the system, which is assumed to be well-defined as in the Boyer-Moore theorem prover [BoMo79]. According to the induction hypotheses, the implementation descriptions in the induction step(s) are immediately replaced by the specifications and the thereby obtained goals are also fed into Verify. If a non-recursive system is to be verified, Verify first translates the temporal abstraction of the specification into a prefix formula using strategy \( \Phi_{PF} \). If additional invariants are given in \( \text{inv}\text{.list} \), these are used by applying strategy \( \Phi_{IV} \) right before applying \( \Phi_{PF} \).

If only a concrete instantiation of the recursion scheme is to be proved (e.g. a 16-bit adder), then Verify translates the data equations into propositional formulas and calls the decision procedure \( \text{HW\_Decide} \) based on symbolic model checking for prefix formulas. If this succeeds, the system has been verified without any interaction. Otherwise, it is checked, if the acceptance condition of the obtained prefix formula has only a safety property. If this is the case, then the prefix formula can be interpreted as a set of universally
The inductionless induction procedure requires some minor interaction for giving appropriate term orderings. In some cases, the inductive completion will run into infinite loops. In these cases, Verify will switch to interactive mode of C@S where further interactions of the user have to be applied to modify the proof goal such that another call to Verify will be more successful.

In general, the more information on the system and its specification (i.e. invariants and lemmas on the abstract data types) is given to the system, the more efficient its verification will be. In the next section, case studies of the common book examples and other systems will show in detail how C@S can be used for the verification of digital systems.

6. Experimental Results

In this section, different circuits are presented to show how different verification strategies are used for their verification. The first case study, a single pulser circuit is verified that has been investigated in [JoMC94] due to its complex temporal behavior. The circuit can be verified automatically by the temporal model checker. The second example, the Black Jack Dealer of the IFIP benchmarks, is also verified automatically by the same strategy though it contains some small parts for processing numerical data. The safe box example illustrates the generation of countermodels. The example is parameterised such that counterexamples of arbitrary length can be generated. Section 6.4 investigates the island tunnel controller presented in [FiJo95] that consists of three parallel processes controlling the restricted access of some agents (the cars) to a common resource (the island). This circuit could also be verified automatically by temporal logic model checking. The same holds also for the arbiter given in section 6.5, that has also been verified by symbolic CTL model checking in [McMi93a]. The generic von Neumann adder given in section 6.6, shows how circuits with abstract data types can be verified by the interactive application of invariant rules. It also shows that the complexity of the verification done by temporal logic model checking can be reduced by applying invariant rules. Note that the invariants express in a formal way the interaction between control and data path of the circuit. The case study given in 6.7 considers sequential n-bit multipliers in order to show that even the use of invariants does not always lead to such an improvement for model checking due to the structure of the invariant. In this case, inductive reasoning with invariant rules has to be used for the proof of additional lemmas in an interactive verification. The systolic arrays verified in section 6.8 do have a simple temporal behaviour, i.e. the specification is a simple safety property. Hence, these examples are verified by term rewriting methods, and as these modules are defined in a generic manner, inductionless induction is used. The
last example given in section 6.9, is a hardware realization of a part of the speech encoding algorithm of the GSM standard.

6.1 Single Pulser

The circuit has a boolean valued input $in$ and a boolean valued output $out$. It is required that between two rising edges of $in$ there is exactly one point of time, where $out$ is high. The circuit is very small, however the specification is not trivial for some formalisms [JoMC94]. Using linear temporal logic, this can be expressed by the following items:

1. $G[\neg in \land Xin \rightarrow X([out \lor (\neg in \land Xin)]) \lor [out \land (\neg in \land Xin)])$
2. $G[out \rightarrow ([X\neg out) \lor (\neg in \land Xin)]$
3. $G[out \rightarrow X\neg out$
4. $G[\neg in \land Xin \rightarrow XFout$

There may be some confusion when the first and the second rising edge occurs. Hence, let us have a closer look at these points. We define a rising edge by the expression $\neg in \land Xin$. This evaluates to true if $in$ is low at the current point of time and high at the following (see figure 6.1). Figure 6.1 indicates that there must be either a single pulse of $out$ (properly) between two rising edges of $in$ or at the time when the second rising edge occurs. In the latter case, it is however mandatory that the next output pulse does not follow immediately. Otherwise we would have the behavior shown in figure 6.1 which has certainly not a single pulse at $out$.

The first specification is satisfied iff after a rising edge at $in$, the output will be high before the next rising edge at $in$ occurs. The second specification requires that each time $out$ holds, it must stay low after that point of time until the next rising edge occurs. If we would only consider these two specifications, then it would be possible to have the behavior given in figure 6.1, since the second line trivially holds when $out$ holds at a point of time when $\neg in \land Xin$ also holds. Hence the third specification explicitly states that $out$ consists of single pulses. The fourth specification demands that $out$ goes high at some time if there has been a rising edge at $in$. 
The runtimes\(^4\), the number of BDD nodes and the number of states for
the verification of the specifications are given in the following table, where
\( C \vdash i \) means the verification problem where specification \( i \) is to be verified:

<table>
<thead>
<tr>
<th>Goal</th>
<th>User Time</th>
<th>BDD nodes</th>
<th>Reach. States</th>
<th>Poss. States</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C \vdash 1 )</td>
<td>0.08</td>
<td>5198</td>
<td>140</td>
<td>( 2^{15} )</td>
</tr>
<tr>
<td>( C \vdash 2 )</td>
<td>0.08</td>
<td>1546</td>
<td>108</td>
<td>( 2^{11} )</td>
</tr>
<tr>
<td>( C \vdash 3 )</td>
<td>0.02</td>
<td>112</td>
<td>14</td>
<td>( 2^5 )</td>
</tr>
<tr>
<td>( C \vdash 4 )</td>
<td>0.05</td>
<td>353</td>
<td>36</td>
<td>( 2^8 )</td>
</tr>
</tbody>
</table>

There are numerous other ways to describe the behavior of the single pulser,
e.g. the following formulae are equivalent to specification 1:

5. \( G [\neg in \land X\bar{in} \rightarrow X[\text{out} \lor [(X\text{out}) \land (\neg \neg in \land X\bar{in})]]] \)

6. \( G [\neg in \land X\bar{in} \rightarrow X[\text{out} \lor [(\neg (\neg \neg in \land X\bar{in})) \lor \text{out}]]] \)

The additional runtimes are given in the following table, where \( C \vdash i \) means
the verification problem where specification \( i \) is to be verified and \( i \equiv j \)
means to problem to verify that the specifications \( i \) and \( j \) are equivalent to
each other.

<table>
<thead>
<tr>
<th>Goal</th>
<th>User Time</th>
<th>BDD nodes</th>
<th>Reach. States</th>
<th>Poss. States</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C \vdash 5 )</td>
<td>0.08</td>
<td>3130</td>
<td>96</td>
<td>( 2^{14} )</td>
</tr>
<tr>
<td>( C \vdash 6 )</td>
<td>0.09</td>
<td>3520</td>
<td>116</td>
<td>( 2^{13} )</td>
</tr>
<tr>
<td>( i = 5 )</td>
<td>0.26</td>
<td>10128</td>
<td>312</td>
<td>( 2^{21} )</td>
</tr>
<tr>
<td>( i = 6 )</td>
<td>0.15</td>
<td>10001</td>
<td>264</td>
<td>( 2^{21} )</td>
</tr>
</tbody>
</table>

The example has no abstract data types at all, but requires to specify and to
verify a quite complex behavior, that can not be done easily in temporal logics
as e.g. in CTL. The specifications we gave are propositional linear temporal
logic formulae that can be translated to prefix formulae as outlined in section
5.3. Note that a part from checking verification problems of the form \( C \vdash i \),
this method allows us also to check the equivalence of specifications (of the
form \( i \equiv j \)).

6.2 Black Jack Dealer

Black Jack is one of the most popular Casino card games. There are at most
seven players and a dealer who deals the cards from a card deck with 52
cards, collects the bets and pays the players. At the beginning, each player
and also the dealer is dealt two cards, one face down and one face up. There
are variations where both cards are face up. Aces may be counted freely
either as 1 or 11. The objective of the game is to reach a score of 21 or to
have a count under 21 but greater than the dealers. If the score is greater
than 21, the player has lost the game. If the dealer goes over 21 the remaining

\[^4\text{All runtimes have been measured on a SUN SPARC10.}\]
players are paid off. If the dealer's cards count sixteen or under the dealer must take a card. If the dealer has a score of seventeen or more he must not take additional cards.

The benchmark circuit (one of the common book examples, see Appendix) plays the dealer's hand in a Black Jack game, i.e. two players are getting cards from a staple of cards and sum up the corresponding values.

The benchmark circuit has a boolean valued input card_rdy and a 4-bit input card. card_rdy signals that the value of a card which is a binary number of \{1, \ldots, 10\} can be read from card. After that the card's value is added to an internal score. It is reasonable to count the first ace as 11 such that the score becomes greater than 16 as soon as possible. Note that it does not make sense to count further aces as 11 since the score would then exceed 21 and the game would be lost. For this reason, the circuit notes if an ace has been added with value 11 to the score. If the score exceeds 21 and an ace has been added with value 11 to the score, the game can still be won if this ace is counted as 1 instead of 11. Hence, in this case the circuit subtracts 10 from the score. If the score has a value greater than 16 but less than 22, the circuit indicates on the output stand that the game is over and that it has won the game. If the score exceeds 21 and no ace has been added with value 11 to the score, the circuit knows that it has lost the game and indicates that at output broke. Finally, if the score is less than 16 the circuit indicates at the output hitme that it demands a further card. The benchmark circuit has an additional reset input for resetting the circuit to its initial state and for clearing the internal score and registers.

\[ r \land \neg (\text{card}_r\text{dy}_s \land \neg \text{card}_r\text{dy}_d) \]

\[ \neg r \lor \neg (\text{score}_\geq 21 \land \text{ace}11) \]

\[ r \land \text{ace} \land \neg \text{ace}11 \]

\[ r \land \text{score}_\geq 21 \land \text{ace}11 \]

Fig. 6.2. State transition diagram of the controller of the Black Jack Dealer
Before presenting the detailed behavior of the benchmark circuit, we first verify that the controller of the Black Jack Dealer (see Appendix) has the state transition diagram shown in figure 6.2. The outputs of the controller are computed by the following equations:

1. set\_broke = test ∧ score\_gt\_21 ∧ ¬ace11
2. clr\_broke = clr\_stand = get ∧ card\_rdy\_s ∧ ¬card\_rdy\_d
3. set\_stand = test ∧ score\_gt\_16 ∧ ¬score\_gt\_21
4. clr\_stand = clr\_broke = get ∧ card\_rdy\_s ∧ ¬card\_rdy\_d
5. set\_ace11 = use
6. clr\_ace11 = (get ∧ card\_rdy\_s ∧ ¬card\_rdy\_d ∧ (stand ∨ broke)) ∨ (test ∧ score\_gt\_21 ∧ ace11)
7. ld\_score = add ∨ use ∨ (test ∧ score\_gt\_21 ∧ ace11)
8. clr\_score = get ∧ card\_rdy\_s ∧ ¬card\_rdy\_d ∧ (stand ∨ broke)
9. adder\_s1 = add
10. adder\_s0 = test ∧ score\_gt\_21 ∧ ace11
11. hitme = get ∧ ¬card\_rdy\_s

Now we can give the detailed behavior of the circuit with the exact timing.

**State get:** This is the initial state and also the state which the circuit is brought to by reset. The circuit is in this state if either the game is over or if it demands a further card. In the first case, its internal score is greater than 16 and it has won the game if and only if the score is less than or equal to 21. This is indicated by the outputs stand and broke, respectively. Otherwise, the circuit indicates at the output hitme that it is ready to accept a new card value. The output functions in this state are as follows:

1. set\_broke = set\_stand = set\_ace11 = ld\_score = 0
2. adder\_s1 = adder\_s0 = 0
3. clr\_broke = clr\_stand = card\_rdy\_s ∧ ¬card\_rdy\_d
4. clr\_ace11 = clr\_score = card\_rdy\_s ∧ ¬card\_rdy\_d ∧ (stand ∨ broke)
5. hitme = ¬card\_rdy\_s

The first line denotes that the values of the internal flipflops and the score remain unchanged in this state, hence, the values of adder\_s1 and adder\_s0 are irrelevant in this state since they are used to choose a value to be added to the score. The signal card\_rdy\_s ∧ ¬card\_rdy\_d is high for one point of time exactly when there was a rising edge at input card\_rdy. The circuit remains in state get until a new card is given. In this case, the circuit switches to state add and clears the flipflops of the data path which hold the values for stand and broke. Note that if the game was not already over, these values are low and hence this has no effect. However, if the game was over, i.e. either stand or broke hold, then the arrival of the new card value starts a new game by also resetting the internal score and the ace11 flag.
For the understanding of the following facts that we have verified, note that \( \neg\text{card}_\text{rdy} \land \text{Xcard}_\text{rdy} \land \text{Xreset} \) is equivalent to \( \text{card}_\text{rdy} \): 

\[
\neg\text{card}_\text{rdy} \land \text{Xcard}_\text{rdy} \land \text{Xreset} \Rightarrow 
\begin{align*}
\text{get} \land \text{reset} \Rightarrow \\
\left( (\text{Xscore} = (\text{[stand} \lor \text{broke}] \Rightarrow 0|\text{score})) \land \\
(\text{Xstand} = 0) \land (\text{Xbroke} = 0) \land \\
(\text{Xace}11 = (\text{[stand} \lor \text{broke}] \Rightarrow 0|\text{ace}11)) \right)
\end{align*}
\]

Get1: \( G[\text{get} \rightarrow \text{X(get} \lor \text{add})] \)

Get2: \( G[\text{get} \rightarrow \text{X(get} \lor \text{add})] \)

Get3: \( G[\neg\text{card}_\text{rdy} \land \text{Xcard}_\text{rdy} \land \text{Xreset} \rightarrow \text{XX}[\text{get} \land \text{reset} \rightarrow \text{Xadd}] \]

Get4: \( G[\text{X} \rightarrow (\text{Xhitme} = \neg(\text{reset} \land \text{card}_\text{rdy}))] \)

**State add:** In this state the circuits's internal score is updated by adding the current card value to the internal score. All outputs are 0 in this state except for the outputs \( \text{ld}_\text{score} \) and \( \text{adder}_s1 \) that command the operation unit to add the card value. If the card is not an ace, the next state will be \( \text{test} \). Otherwise it is checked if this is the first ace. If this is the case, the next state is state \( \text{use} \) which is used to set the \( \text{ace}11 \) flag and to add 10 to the score, i.e. the first ace is counted as 11. On the other hand, if an ace has already been added to the score, the currently read ace must be counted as 1 and the next state is hence \( \text{test} \).

Add1: \( G[\text{add} \land \text{reset} \rightarrow \left( (\text{Xscore} = (\text{score} + \text{card})) \land \\
(\text{Xstand} = 0) \land (\text{Xbroke} = 0) \land \\
(\text{Xace}11 = \text{ace}11) \right)] \)

Add2: \( G[\text{add} \land \text{reset} \rightarrow \text{X(use} \lor \text{test})] \)

Add3: \( G[\text{add} \land \text{reset} \land \text{acecard} \land \neg\text{ace}11 = \text{Xuse}] \)

Add4: \( G[\text{add} \land \text{reset} \land \neg(\text{acecard} \land \neg\text{ace}11) \rightarrow \text{Xtest}] \)

Add5: \( G[\text{add} \rightarrow \neg(16 \leq \text{score})] \)

**State use:** In this state all outputs of the controller are false except for \( \text{set}_a\text{ce}11, \text{ld}_\text{score} \). Hence, in this state the \( \text{ace}11 \) flag is set and 10 is added to the internal score as already mentioned.

Use1: \( G[\text{use} \land \text{reset} \rightarrow \left( (\text{Xscore} = (\text{score} + \text{card})) \land \\
(\text{Xstand} = 0) \land (\text{Xbroke} = 0) \land \\
(\text{Xace}11 = 1) \right)] \)

Use2: \( G[\text{use} \land \text{reset} \rightarrow \text{Xtest}] \)

Use3: \( G[\text{use} \rightarrow \neg(17 \leq \text{score})] \)

Use4: \( G[\text{use} \land \text{reset} \rightarrow \neg\text{ace}11 \land \neg\text{stand} \land \neg\text{broke} \land \neg\text{hitme}] \)

**State test:** In this state the internal score of the circuits data path is compared with 16 and 21 in order to find out whether the game is over or not. The output functions are as follows:

- \( \text{set}_\text{broke} = \text{score}_\text{gt}_\text{21} \land \neg\text{ace}11 \)
- \( \text{clr}_\text{broke} = \text{clr}_\text{stand} = \text{set}_\text{ace}11 = \text{clr}_\text{score} = \text{hitme} = \text{adder}_s1 = 0 \)
- \( \text{set}_\text{stand} = \text{score}_\text{gt}_\text{16} \land \neg\text{score}_\text{gt}_\text{21} \)
- \( \text{clr}_\text{ace}11 = \text{ld}_\text{score} = \text{score}_\text{gt}_\text{21} \land \text{ace}11 \)

Note that \( \text{score}_\text{gt}_\text{21} \) implies \( \text{score}_\text{gt}_\text{16} \). There are three cases to distinguish in this state. In the first case, \( \text{score} \leq 16 \) holds though eventually
a (the first) ace has been added with value 11. In this case, the game is neither won or lost for the circuit, the flipflops of the data path remain unchanged and the next state is \( \text{get} \). In the second case, \( 16 < \text{score} \leq 21 \) holds. In this case, the game is over for the dealer. This has the effect that the \( \text{stand} \) flag is set and the next state is also \( \text{get} \). Finally, in the third case \( 21 < \text{score} \) holds. Then it is checked if an ace has been added with value 11 to the score. If this is the case then 10 is subtracted from the score such that this ace is counted only with 1 and the \( \text{ace11} \) flag is cleared. The next state will then also be \( \text{test} \) since it must then again be checked whether the game is over or not. Note however that the loop from state \( \text{test} \) to \( \text{test} \) can occur at most once since this transition clears the \( \text{ace11} \) flag which must be set for the transition.

\[
\begin{align*}
\text{Test1 G} & \quad \left( \text{test} \land \text{reset} \rightarrow \left( \left[ \left( 21 < \text{score} \right) \land \text{ace11} \right] \rightarrow \Box \text{score} = (\text{score} - 10) \right) \land \\
& \quad \left( \lnot \left[ \left( 21 < \text{score} \right) \land \text{ace11} \right] \rightarrow \Box \text{score} = \text{score} \right) \land \\
& \quad \left( \Box \text{stand} = \left( 16 < \text{score} \right) \land \lnot \left( 21 < \text{score} \right) \right) \land \\
& \quad \left( \Box \text{broke} = \left( 21 < \text{score} \right) \land \lnot \text{ace11} \right) \land \\
& \quad \left( \text{ace11} = \lnot \left( \left( 21 < \text{score} \right) \land \text{ace11} \right) \land \text{ace11} \right) \right)
\end{align*}
\]

\[
\begin{align*}
\text{Test2 G}[\text{test} \land \text{reset} \rightarrow X(\text{get} \lor \text{test})] \\
\text{Test3 G}[\text{test} \land \text{reset} \land \left( 21 < \text{score} \right) \land \text{ace11} \rightarrow X \text{test}] \\
\text{Test4 G}[\text{test} \land \text{reset} \land \lnot \left( \left( 21 < \text{score} \right) \land \text{ace11} \right) \rightarrow X \text{get}] \\
\text{Test5 G}[\text{test} \land \text{reset} \rightarrow X(\text{test} \rightarrow X \text{get})]
\end{align*}
\]

Beneath the specifications that have to hold in the specific controller states, there are also global specification that have to hold for all states:

\[
\begin{align*}
\text{Global1} & \quad \left( 21 < \text{score} \right) \rightarrow \left( 16 < \text{score} \right) \\
\text{Global2} & \quad \text{get} \land X \text{get} \land \text{score} = 0 \land \lnot \text{stand} \land \lnot \text{broke} \land \lnot \text{ace11} \land \text{hitme} \\
\text{Global3} & \quad G[\lnot \text{reset} \rightarrow X(\text{get} \land \lnot \text{stand} \land \lnot \text{broke} \land \lnot \text{ace11} \land \text{score} = 0)] \\
\text{Global4} & \quad [G \text{Fcard\_rdy}] \rightarrow [G \text{Fget}] \\
\text{Global5} & \quad [\text{Greset}] \land [G F(\lnot \text{card\_rdy} \land X \text{card\_rdy})] \rightarrow [G \text{Fadd}] \\
\text{Global6} & \quad [\text{Greset}] \land [G F(\lnot \text{card\_rdy} \land X \text{card\_rdy})] \rightarrow [G \text{Fuse}] \\
\text{Global7} & \quad [G \text{reset}] \land [G F(\lnot \text{card\_rdy} \land X \text{card\_rdy})] \land [G (\text{card} \neq 0)] \rightarrow [G F(\text{stand} \lor \text{broke})]
\end{align*}
\]

The runtimes of all results are given table 6.1 where in each case the data path has been reduced to propositional logic with strategy \( \Theta_B \) and the remaining specification has been translated to a prefix formula by strategy \( \Phi_{PF} \) without any application of an invariant rule or other interactions. From the viewpoint of data handling, the example is a very simple one, but not all the specifications can be expressed in CTL (e.g. Global7). Hence, from a viewpoint of temporal reasoning, the example is not as trivial.

### 6.3 Safe Box

This example is due to William Keisler and is based on the Chinese Ring Puzzle. Assume there is a safe box that has \( n \) knobs \( k_{n-1}, \ldots, k_0 \), such that
<table>
<thead>
<tr>
<th>Goal</th>
<th>Runtime [sec]</th>
<th>BDD nodes</th>
<th>Trans. Relation</th>
<th>Reach. States/Poss. States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get1</td>
<td>0.83</td>
<td>11335</td>
<td>2207</td>
<td>77632/2^{30}</td>
</tr>
<tr>
<td></td>
<td>3.36</td>
<td>36649</td>
<td>14064</td>
<td>179008/2^{24}</td>
</tr>
<tr>
<td>Get2</td>
<td>0.36</td>
<td>10083</td>
<td>1181</td>
<td>41792/2^{20}</td>
</tr>
<tr>
<td>Get3</td>
<td>0.66</td>
<td>10197</td>
<td>1761</td>
<td>68032/2^{27}</td>
</tr>
<tr>
<td>Get4</td>
<td>0.39</td>
<td>10578</td>
<td>1102</td>
<td>31168/2^{21}</td>
</tr>
<tr>
<td>Add1</td>
<td>2.00</td>
<td>21455</td>
<td>9540</td>
<td>353920/2^{27}</td>
</tr>
<tr>
<td>Add2</td>
<td>0.37</td>
<td>10561</td>
<td>1113</td>
<td>37824/2^{21}</td>
</tr>
<tr>
<td>Add3</td>
<td>0.46</td>
<td>11227</td>
<td>1270</td>
<td>60224/2^{23}</td>
</tr>
<tr>
<td>Add4</td>
<td>0.46</td>
<td>11227</td>
<td>1270</td>
<td>60224/2^{23}</td>
</tr>
<tr>
<td>Add5</td>
<td>0.37</td>
<td>10949</td>
<td>1126</td>
<td>30976/2^{18}</td>
</tr>
<tr>
<td>Use1</td>
<td>1.24</td>
<td>22376</td>
<td>7835</td>
<td>115328/2^{26}</td>
</tr>
<tr>
<td>Use2</td>
<td>0.37</td>
<td>10182</td>
<td>1117</td>
<td>35264/2^{21}</td>
</tr>
<tr>
<td>Use3</td>
<td>0.36</td>
<td>10949</td>
<td>1126</td>
<td>30976/2^{18}</td>
</tr>
<tr>
<td>Use4</td>
<td>0.36</td>
<td>10699</td>
<td>1093</td>
<td>30976/2^{18}</td>
</tr>
<tr>
<td>Test1</td>
<td>1.34</td>
<td>20188</td>
<td>9243</td>
<td>129792/2^{28}</td>
</tr>
<tr>
<td></td>
<td>1.36</td>
<td>20210</td>
<td>9245</td>
<td>129792/2^{28}</td>
</tr>
<tr>
<td></td>
<td>0.60</td>
<td>12040</td>
<td>1603</td>
<td>49920/2^{27}</td>
</tr>
<tr>
<td>Test2</td>
<td>0.37</td>
<td>10004</td>
<td>1153</td>
<td>44480/2^{27}</td>
</tr>
<tr>
<td>Test3</td>
<td>0.52</td>
<td>12615</td>
<td>1314</td>
<td>49152/2^{23}</td>
</tr>
<tr>
<td>Test4</td>
<td>0.49</td>
<td>12615</td>
<td>1314</td>
<td>49152/2^{23}</td>
</tr>
<tr>
<td>Test5</td>
<td>0.53</td>
<td>10077</td>
<td>1284</td>
<td>61312/2^{24}</td>
</tr>
<tr>
<td>Global1</td>
<td>0.35</td>
<td>10949</td>
<td>1126</td>
<td>30976/2^{18}</td>
</tr>
<tr>
<td>Global2</td>
<td>2.02</td>
<td>24900</td>
<td>10702</td>
<td>132480/2^{30}</td>
</tr>
<tr>
<td>Global3</td>
<td>0.41</td>
<td>10510</td>
<td>1097</td>
<td>31104/2^{20}</td>
</tr>
<tr>
<td>Global4</td>
<td>0.52</td>
<td>10907</td>
<td>1158</td>
<td>930048/2^{24}</td>
</tr>
<tr>
<td>Global5</td>
<td>0.71</td>
<td>10818</td>
<td>1392</td>
<td>1262850/2^{28}</td>
</tr>
<tr>
<td>Global6</td>
<td>0.69</td>
<td>10524</td>
<td>1437</td>
<td>1482500/2^{28}</td>
</tr>
<tr>
<td>Global7</td>
<td>9.10</td>
<td>15843</td>
<td>3130</td>
<td>15231200/2^{33}</td>
</tr>
</tbody>
</table>

Table 6.1. Runtime data for the verification of the Black Jack Dealer

Each knob $k_i$ has only two possible positions: open (0) or close (1). The knobs cannot be turned independently:

1. Initially, all knobs are in their closed position.
2. The leftmost knob $k_{n-1}$ can always be turned.
3. If one doesn't choose to turn $k_{n-1}$, then the only knob that can be turned is the one directly to the right of the first closed knob from the left.
4. If the last knob $k_0$ is the only one in the closed position then the above rule does not apply, and the only choice left is to turn $k_{n-1}$.
5. At each point of time, only one knob can be turned.

The objective is to open the box, i.e. to set all knobs to their open position by a sequence of knob turns. The example can be used as benchmark especially for testing the generation of countermodels, if we state that the safe box
can not be opened at all. The verification system must then disprove the statement by finding a knob pressing sequence that opens the safe. We will see that the minimal length \( \ell(n) \) of the safe box problem with \( n \) knobs will have a length in \( O(2^n) \), hence the length of the countermodel will grow rapidly.

A knob \( k_i \) is pressed at a certain point of time iff \( Xk_i = \neg k_i \) holds for that point of time. The following facts can be proved for the safe opener problem:

**Lemma 6.1.** The \( n \)-knobs safe box problem is given by the following state transition equations:

\[
\left( \bigwedge_{i=0}^{n-1} (k_i = 1) \right) \land \left( \bigwedge_{i=0}^{n-2} G(Xk_i = \neg k_i = k_{i+1} \land \bigwedge_{j=i+2}^{n-1} \neg k_j) \right)
\]

The following facts hold for the corresponding state decision diagram:

1. Each state has at most two successor states: one of them is reached by pressing the leftmost knob \( k_{n-1} \) and the other one is reached by pressing the knob \( k_i \) for which \( k_{i+1} \land \bigwedge_{j=i+2}^{n-1} \neg k_j \) holds if one such knob exists.
2. The only two states that have only one successor state are the states \((0, \ldots, 0)\) and \((0, \ldots, 0, 1)\).
3. If a state \( s_1 \) is the successor of a state \( s_0 \) then \( s_0 \) is also a successor of \( s_1 \).
4. The minimal length of a knob pressing sequence to open the \( n \)-knob safe is:

\[
\ell(n) := \begin{cases} 
\frac{2}{3}(2^n - \frac{1}{2}) & : \text{if } n \text{ is odd} \\
\frac{2}{3}(2^n - 1) & : \text{if } n \text{ is even}
\end{cases}
\]

**Proof.** To prove 1. let us check that the second possibility for pressing a knob is uniquely defined if it exists. To see this, assume there are two such knobs \( k_i \) and \( k_p \) with that property. Without loss of generality assume \( i < p \). This leads immediately to a contradiction as the property for knob \( k_i \) demands that \( \neg k_{p+1} \) holds. To prove 2., note that if in a state only the leftmost knob \( k_{n-1} \) can be pressed, this means that for all \( i \in \{0, \ldots, n-2\} \) the knob \( k_i \) can not be pressed which in turn means that \( k_{i+1} \land \bigwedge_{j=i+2}^{n-1} \neg k_j \) does not hold for all \( i \in \{0, \ldots, n-2\} \). Hence, all states who have only one successor are solutions of the following formula:

\[
\neg k_1 \lor k_2 \lor k_3 \lor \ldots \lor k_{n-1} \\
\land \neg k_2 \lor k_3 \lor \ldots \lor k_{n-1} \\
\land \neg k_3 \lor \ldots \lor k_{n-1} \\
\vdots \\
\land \neg k_{n-2} \lor k_{n-1} \\
\land \neg k_{n-1}
\]

It can be easily seen that there are only two solutions: as all rows have to be fulfilled, the last row requires that \( \neg k_{n-1} \) has to hold. This implies...
that \( \neg k_{n-2} \) must hold, and in turn that \( \neg k_{n-3} \) must hold and so on. Hence, the above formula is equivalent to \( \bigwedge_{j=1}^{n-1} \neg k_j \), and hence the states which do only have one successor are the states \((0, \ldots, 0)\) and \((0, \ldots, 0, 1)\). Hence, if \( s \) is the number of reachable states, then the number of edges is exactly \( 2(s-1) \). The third proposition is quite simple to prove. Assume \( s_1 \) is reached from \( s_0 \) by pressing the leftmost knob \( k_{n-1} \). Then \( s_0 \) is reached by pressing the leftmost knob \( k_{n-1} \) again in state \( s_1 \). If on the other hand the second possibility is chosen, then it has to be noticed that the knob that has been pressed can be pressed also in the successor state because the decision whether a knob \( k_i \) may be pressed or not only depends on the state of the knobs \( k_{i+1}, \ldots, k_{n-1} \). Pressing the knob \( k_i \) does however not affect the state of the knobs \( k_{i+1}, \ldots, k_{n-1} \).

The propositions 1-3 show that the state transition diagram consists of two 'chains' of states, one ending in \((0, \ldots, 0, 1)\) and the other one ending in \((0, \ldots, 0)\) (see figure 6.3 for an example). The latter one opens the box and is roughly 2 times longer than the other one. Let \( \ell_0(n) \) and \( \ell_1(n) \) be the lengths of the chains ending in \((0, \ldots, 0)\) and \((0, \ldots, 0, 1)\), respectively. Then, a detailed analysis shows that

\[
\ell_0(n) := \begin{cases} 
2\ell_0(n-1) + 1 & : \text{if } n \text{ is odd} \\
2\ell_0(n-1) & : \text{if } n \text{ is even}
\end{cases}
\]

holds. By induction on \( n \) it follows that \( \ell_0(n) + \ell_0(n-1) = 2^{n-1} \) holds. This in turn implies \( \ell_0(n) - \ell_0(n-2) = (\ell_0(n) + \ell_0(n-1)) - (\ell_0(n-1) + \ell_0(n-2)) = (2^n - 1) - (2^{n-1} - 1) = 2^{n-1}, \) hence \( \ell_0(n) = 2^{n+1} + \ell_0(n-2) \). Now it is easy to see that \( \ell_0(2n+1) = \sum_{i=0}^{n} 2^{2i} \) and \( \ell_0(2n) = \sum_{i=0}^{n-1} 2^{2i+1} \). The final result
follows from the general equation for geometric sums:

\[ \sum_{i=0}^{n} q^i = \frac{q^{n+1} - 1}{q - 1} \]

Figure 6.3 shows the state transition diagram for the safe box problem of a safe with 4 knobs. The state transition diagram shown in figure 6.3 is typical for all \( n \geq 3 \): two chains of states are starting in the initial state. One chain ‘ends’ in the state \((0, \ldots, 0)\) and the other one ‘ends’ in the state \((0, \ldots, 0, 1)\). According to the previously proved facts, there are no more than two chains
as the only possible end points are the states \((0, \ldots, 0)\) and \((0, \ldots, 0, 1)\). The previous lemma also shows that for each number \(n\) there is a knob pressing sequence that opens the safe with \(n\) knobs in roughly \(\frac{3}{2}2^n\) steps.

The transition equations given in the previous lemma can be implemented as a synchronous circuit that consists of \(n\) cells for the \(n\)-knob problem. Each cell has inputs \(l_{\text{node}}, l_{\text{conj.in}}\) and the outputs \(k_i\) and \(l_{\text{conj.out}}\), which are defined as

\[
\begin{align*}
- k_i &= 1 \land G[Xk_i = -k_i = l_{\text{node}} \land l_{\text{conj.in}}] \\
- l_{\text{conj.out}} &= -l_{\text{node}} \land l_{\text{conj.in}}
\end{align*}
\]

The cells are connected as shown in figure 6.4. Note that the leftmost knob \(k_{n-1}\) is an input for the circuit and only the other positions of the knobs \(k_i\) are computed. Hence, the circuit can actually press two knobs at a certain point of time. It can also be seen that (consider proposition 3 of the previous lemma) in the minimal opening sequence alternatively the leftmost knob and another knob is pressed. The circuit can perform these two steps at once and can thus halves the number of operations. The specification which is to be disproved is simply \(G \left( \bigwedge_{i=0}^{n-1} k_i \right)\).

<table>
<thead>
<tr>
<th>knobs</th>
<th>(\ell(\text{knobs}))</th>
<th>states in countermodel</th>
<th>Runtime [sec]</th>
<th>BDD nodes</th>
<th>Trans-Rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0.03</td>
<td>259</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
<td>0.05</td>
<td>360</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>8</td>
<td>0.08</td>
<td>864</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>11</td>
<td>0.08</td>
<td>1395</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>24</td>
<td>0.20</td>
<td>2895</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>85</td>
<td>43</td>
<td>0.36</td>
<td>5426</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>170</td>
<td>88</td>
<td>0.72</td>
<td>10006</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>341</td>
<td>171</td>
<td>1.72</td>
<td>10004</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>682</td>
<td>344</td>
<td>3.97</td>
<td>13621</td>
<td>82</td>
</tr>
<tr>
<td>11</td>
<td>1365</td>
<td>683</td>
<td>8.98</td>
<td>26622</td>
<td>92</td>
</tr>
<tr>
<td>12</td>
<td>2730</td>
<td>1368</td>
<td>18.98</td>
<td>52762</td>
<td>102</td>
</tr>
<tr>
<td>13</td>
<td>5461</td>
<td>2731</td>
<td>44.28</td>
<td>104670</td>
<td>112</td>
</tr>
<tr>
<td>14</td>
<td>10922</td>
<td>5464</td>
<td>104.1</td>
<td>208650</td>
<td>122</td>
</tr>
<tr>
<td>15</td>
<td>21845</td>
<td>10923</td>
<td>277.36</td>
<td>416245</td>
<td>132</td>
</tr>
</tbody>
</table>

Table 6.2. Results for the safe box opening problem

The runtimes and the lengths of the countermodel we obtained are given in table 6.2. As can be seen, the lengths of the countermodels are almost optimal, i.e. half of \(\ell(\text{knobs})\) (as the circuit can perform these two steps at once). However, we have to admit that this is due to the very good implementation of SMV that is used as backend model checker of our LTL model checker. The runtimes and the storage requirements, however seem to grow exponentially such that about 39.6 MBytes of main memory are required to
find the countermodel for the 15 knobs problem. We have only investigated for the automatic verification of the safe box opening problem. A machine checked proof of lemma 6.1 has not been done with C@S, but should be possible. However, we assume that it would require a considerable amount of user interactions.

### 6.4 The Island Tunnel Controller

This controller problem is an extension of the *traffic light control* problem and, as such it is a metaphor for coordinating access to a restricted resource [FiJo95]. Concurrent entry/exit is allowed under certain conditions, represented by a one-lane tunnel between the mainland and the island (see figure 6.5) that may at any time contain several vehicles, and an additional constraint on the number of vehicles that can be on the island. The circuit has the task to control the traffic lights at the each end of this tunnel. Of course, the specification requires that both traffic lights should never have a green light at the same time. Moreover, the system is always alive, i.e. if a car wants to enter the tunnel, it may do so after a finite amount of time. Furthermore, the traffic on the island is limited: at no point of time there should be more than 16 cars on the island.

![Traffic Light Control Problem](image)

**Fig. 6.5. Traffic Light Control Problem**

There are four sensors for detecting the presence of vehicles: one at the tunnel entrance on the island side \( il_{\text{enter}} \), one at the tunnel exit on the island side \( il_{\text{exit}} \), one at the tunnel entrance on the mainland side \( ml_{\text{enter}} \), and one at the tunnel exit on the mainland side \( ml_{\text{exit}} \). The controller has two counters \( IC \) and \( TC \), the latter for counting cars inside the tunnel, the other one for
counting cars currently on the island or inside the tunnel traveling to the island. The design assumes the following assumptions on its environment, described in terms of the behavior of vehicles that entering the system:

A1: Cars are not produced in the tunnel, i.e. if there is no car in the tunnel, then no car can exit the tunnel (on none of its sides) at this point of time.
A2: Cars do not disappear in the tunnel.
A3: Cars are not produced in the island.
A4: The island counter counts the cars which are on the island or which are currently inside the tunnel traveling to the island. If all cars on the island are currently inside the tunnel then no car can enter the tunnel at the island side.
A5: Each car wants to leave the island after some time.
A6: If the island traffic light is green, then cars can only exit the tunnel at the mainland side until the mainland traffic light turns to green (analogously for the mainland).
A7: If a car wants to enter a side of the tunnel where the traffic light is green, it will actually enter the tunnel after some time.

The above assumptions can be expressed as follows in temporal logic:

A1: \[G[(TC = 0) \rightarrow \neg il\_exit \land \neg ml\_exit]\]
A2: \[G[(TC \neq 0) \rightarrow F[il\_exit \lor ml\_exit]]\]
A3: \[G[(IC = 0) \rightarrow \neg il\_enter]\]
A4: \[G[(IC = TC) \rightarrow \neg il\_enter]\]
A5: \[G[(IC \neq 0) \rightarrow il\_enter]\]
A6: \[G\left[ (il\_green \rightarrow [(\neg il\_exit) \lor ml\_green]) \land \\
                 (ml\_green \rightarrow [(\neg ml\_exit) \lor il\_green]) \right]\]
A7: \[G\left[ (il\_enter \land il\_green \land Xil\_green \rightarrow F \neg il\_enter) \land \\
                 (ml\_enter \land ml\_green \land Xml\_green \rightarrow F \neg ml\_enter) \right]\]

Some of the specifications depend on the fact that only a maximal number of cars is allowed to be on the island or depend on the number of cars in the tunnel, others do not. The latter ones can be proved without the counters and hence, for any number of maximal cars. This implies that some of the following specifications require only a subset of the above assumptions, and some even none of them.

S1: At no point of time, both lights are green.
S2: At no point of time, there are more than 16 cars on the island.
S3: If a car wants to enter the tunnel and persists to enter the tunnel, then it has the chance to do so after a finite time.
S4: Traffic lights can change only when the tunnel is empty.

The formalizations of the above specifications are as follows:

S1: \[G \neg [ml\_green \land il\_green]\]
S2: $G[IC \leq 16]$

S3: $G[\neg G[il\_enter \land il\_red] \land \neg G[ml\_enter \land ml\_red]]$

S4: $G\left[(ml\_red \land Xml\_green \rightarrow (TC = 0)) \land (il\_red \land Xil\_green \rightarrow (TC = 0))\right]$

The specification S3 can also be expressed equivalently as follows: If a car wants to enter a side of the tunnel where the light is red and the car waits until the light changes, then the light will actually change after some time. Hence, the formula $G[\neg G[enter \land red]]$ is equivalent to

$$G[enter \land red \land [(enter \land red) \lor (enter \land green)] \rightarrow F(enter \land green)]$$

![Diagram](image)

**Fig. 6.6.** Three-process architecture of the Island Tunnel Control problem.

An implementation of the tunnel controller was given in [FiJo95] and involves three processes implemented by three subcontrollers (figure 6.6) and two counters for counting the number of cars presently inside the tunnel (TC) and the number of cars presently on the island (IC). As subcontrollers there are two side controllers and a tunnel access controller. The latter is an arbiter which grants access to the tunnel either for the mainland or for the island controller. The side controllers behave equivalent, their inputs and outputs have the following meaning (each signal is prefixed later on by either *il* or *ml*):

- *green* and *red* are the outputs for generating green and red lights for the particular side.
- *use* indicates that this side is currently allowed to use the tunnel.
- `req` indicates that the controller has currently no access to the tunnel but wants to access it.
- `yield` indicates that the controller is being instructed to release control of the tunnel.
- `grant` indicates that the island has been granted control of the tunnel.
- `tc_inc` indicates that tunnel controller has to be incremented.
- `tc_dec` indicates that tunnel controller has to be decremented.
- `ic_change` indicates that island controller has to be modified (if the signal comes from the island side it has to be decremented, otherwise it has to be incremented).

![State transition diagram for the island/mainland controller](image)

**Fig. 6.7.** State transition diagram for the island/mainland controller

The behaviour of the side controller is given in figure 6.7, where the reset behaviour is neglected for reasons of simplicity. The output equations of the side controllers are as follows:

- `green_light := green V entering`
- `red_light := red V exiting`
- `use := green V entering`
- `req := red V enter`
- `tc_inc := green V ~yield V res_level V enter`
- `tc_dec := red V exit`
- `ic_change := green V ~yield V res_level V enter`

Initially, the side controllers are in state `red` and do not have access to the tunnel, both counters are zero, and there are no vehicles neither on the island nor in the tunnel. Cars are not allowed to enter the tunnel in this state as
a red light is produced, but cars may exit the tunnel. Each time a car exits the tunnel at this side, this is detected via `exit` and the tunnel counter is decremented. While the car is exiting the tunnel, the controller is in the exiting state where also a red light is produced. After the car has exited the tunnel, the controller will be again in the red state. If a car wants to enter the tunnel in this state, the signal `enter` will indicate this and the controller signals via `req` to the tunnel access controller that it wants to access the tunnel. If the other side controller has the control over the tunnel, the tunnel access controller will instruct the other side controller to release it and will then wait until no more cars are inside the tunnel. After that, the tunnel access controller will allow access to the tunnel and the side controller switches to its green state. In this state, a green light is produced and cars may enter the tunnel. Each time the car enters the tunnel, the tunnel counter is incremented and the island counter is modified, i.e. incremented if we are on the mainland and decremented otherwise. However, if the side controller is on the mainland side, it has furthermore to check if more than 16 cars are on the island. This is checked via the input `res_level` which is always high on the island side controller and equals to $IC < 16$ on the mainland side. If the limit of 16 cars is reached, the controller switches automatically to its red state.

The behaviour of the tunnel access controller is given in figure 6.8. The initial state is `dispatch`, where no side controller has the control over the tunnel. Again, for reasons of simplicity, the reset behaviour is suppressed in figure 6.8, but it has to be noted that the controller always switches to state `dispatch` when `reset` becomes high. The output equations of the tunnel access controller are as follows:

- $il\_grant := (dispatch \land il\_req \land \neg ml\_use \land (TC = 0)) \lor (ml\_clear \land (TC = 0))$
- $ml\_grant := (dispatch \land \neg il\_req \land ml\_req \land (IC < 16)) \land \neg il\_use \land (TC = 0)) \lor (il\_clear \land (TC = 0))$
- $il\_yield := il\_use$
- $ml\_yield := ml\_use$

The states have the following meaning: the controller is in state `ml\_clear` when the island controller wants to access the tunnel and the mainland controller does not use the tunnel, but there are still cars inside the tunnel (from a previous use by the mainland controller). `ml\_use` is the state, where the mainland controller has control over the tunnel, but the island controller wants to access it. The states `il\_clear` and `il\_use` are used analogously.

The island controller has a higher priority than the mainland controller. If neither the island nor the mainland controller have control over the tunnel, and both request it at the same point of time, then there are two cases: if there are no cars in the tunnel, the island controller is granted immediately access to the tunnel. Otherwise, the tunnel access controller switches to state
Fig. 6.8. State transition diagram for the tunnel access controller

mlclear where it remains until all cars have left the tunnel. Then there are no more cars in the tunnel and the island controller is granted access to it.

If, on the other hand, the island controller requests the control over the tunnel and the tunnel is currently used by the mainland controller, then the tunnel access controller switches to state mluse and sends the yield signal to the mainland controller, i.e. instructs it to release the control over the tunnel. The mainland controller will do so either immediately or, if currently a car is entering at the mainland side, after this car has entered. The mainland controller signals the release of the tunnel by the signal mluse, which forces the tunnel access controller to switch to state mclear, where it remains until all cars have left the tunnel. Then it switches again to state dispatch and grants access of the tunnel to the island controller. The island controller switches then to state green and outputs via iluse that it has now taken control over the tunnel.

Now, assume the mainland controller wants to access the tunnel, but the island controller does not. Then some more cases are considered:

1. If the island counter equals to 16, then the access is not granted.
2. If the island counter is less than 16 and the tunnel is currently not used by the island controller, then the tunnel access controller switches to state
ilclear to empty the tunnel and grants access to the mainland controller after the tunnel is emptied.

3. If the island counter is less than 16 and the tunnel is currently used by the island controller (though it does not request it at the moment), then the tunnel access controller switches to state iluse, sends the yield signal to the island controller. After the island controller has released the tunnel, the tunnel is emptied and the access is granted to the mainland controller.

The island tunnel controller can be implemented for arbitrary numbers of cars that may be on the island, hence, it is a generic problem where the number of cars is a parameter. In [FiJo95], the problem has been presented for 16 cars as in this chapter. Table 6.3 lists our verification results for some other numbers of cars. A generic verification of the problem has not been done; but we suppose that it should be possible without too much interactions.

6.5 Arbiter

The task of an arbitration circuit is to manage requests of components for a shared resource like a bus. The arbitration circuit obtains all requests and give at each point of time at most one component a grant for the common resource. The circuit that is considered in this section is due to David Dill and can also be found in [McMi93a]. The circuit can be implemented for an arbitrary number of components and has hence \( n \) inputs \( req_0, \ldots, req_{n-1} \) where each component can request a grant for the shared resource. The circuit has also \( n \) outputs \( ack_0, \ldots, ack_{n-1} \) for allowing access to one of the components. The implementation of the arbitration circuit of this section assumes that each access to the component does not take longer than one point of time. Hence, at each point of time a new component may access the resource. Furthermore, it is assumed that the component holds its access request until it either obtains the access or it does not need it any more. The implementation of the arbitration circuit is given in figure 6.9.

The implementation for \( n \) components consists of \( n \) cells, which are all implemented in the same manner, except for the first one. This is due to the fact, that the arbiter considers priorities as well as a round-robin scheme: there is a token which circulates through the cells of the arbiter. Hence, the initialisation of the cells requires that initially one cell obtains the token and the others do not have a token at that point of time. Beneath the token flag, each cell has another flag (the persistence flag) that is set when the corresponding component wants to access the resource, but does not achieve it though its cell has the token. This gives the cell dynamically a higher priority and it can definitely access the resource when the token arrives again in that cell. If the cell with the token has however not set the persistence flag, the access is computed by a static priority: the component with the lowest index will then receive the access.
<table>
<thead>
<tr>
<th>spec</th>
<th>cars</th>
<th>Runtime [seconds]</th>
<th>BDD nodes</th>
<th>Trans-Relation</th>
<th>Reachable States</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2</td>
<td>1.30</td>
<td>13080</td>
<td>2420</td>
<td>13472/2^{22}</td>
</tr>
<tr>
<td>S2</td>
<td>2</td>
<td>2.51</td>
<td>15880</td>
<td>2600</td>
<td>24224/2^{23}</td>
</tr>
<tr>
<td>S3.il</td>
<td>2</td>
<td>2.85</td>
<td>16728</td>
<td>2836</td>
<td>33376/2^{25}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>2</td>
<td>2.54</td>
<td>15532</td>
<td>2630</td>
<td>35936/2^{25}</td>
</tr>
<tr>
<td>S1</td>
<td>3</td>
<td>1.35</td>
<td>13838</td>
<td>2227</td>
<td>15872/2^{22}</td>
</tr>
<tr>
<td>S2</td>
<td>3</td>
<td>1.67</td>
<td>13141</td>
<td>2235</td>
<td>15872/2^{23}</td>
</tr>
<tr>
<td>S3.il</td>
<td>3</td>
<td>3.32</td>
<td>14925</td>
<td>2821</td>
<td>40928/2^{25}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>3</td>
<td>2.79</td>
<td>16053</td>
<td>2470</td>
<td>43040/2^{25}</td>
</tr>
<tr>
<td>S1</td>
<td>4</td>
<td>3.57</td>
<td>18720</td>
<td>3362</td>
<td>45216/2^{24}</td>
</tr>
<tr>
<td>S2</td>
<td>4</td>
<td>6.84</td>
<td>19266</td>
<td>3576</td>
<td>83104/2^{25}</td>
</tr>
<tr>
<td>S3.il</td>
<td>4</td>
<td>8.21</td>
<td>17184</td>
<td>4154</td>
<td>112736/2^{27}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>4</td>
<td>7.54</td>
<td>18060</td>
<td>3636</td>
<td>121696/2^{27}</td>
</tr>
<tr>
<td>S1</td>
<td>5</td>
<td>3.82</td>
<td>21547</td>
<td>3414</td>
<td>52384/2^{24}</td>
</tr>
<tr>
<td>S2</td>
<td>5</td>
<td>7.35</td>
<td>17479</td>
<td>3634</td>
<td>98464/2^{25}</td>
</tr>
<tr>
<td>S3.il</td>
<td>5</td>
<td>8.84</td>
<td>18154</td>
<td>4292</td>
<td>132384/2^{27}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>5</td>
<td>8.13</td>
<td>19563</td>
<td>3704</td>
<td>140640/2^{27}</td>
</tr>
<tr>
<td>S1</td>
<td>6</td>
<td>3.94</td>
<td>19212</td>
<td>3315</td>
<td>59552/2^{24}</td>
</tr>
<tr>
<td>S2</td>
<td>6</td>
<td>7.75</td>
<td>19657</td>
<td>3582</td>
<td>113824/2^{26}</td>
</tr>
<tr>
<td>S3.il</td>
<td>6</td>
<td>9.68</td>
<td>18212</td>
<td>4301</td>
<td>152160/2^{27}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>6</td>
<td>8.71</td>
<td>19865</td>
<td>3647</td>
<td>159584/2^{27}</td>
</tr>
<tr>
<td>S1</td>
<td>7</td>
<td>4.07</td>
<td>19631</td>
<td>2934</td>
<td>64512/2^{24}</td>
</tr>
<tr>
<td>S2</td>
<td>7</td>
<td>4.71</td>
<td>17965</td>
<td>2942</td>
<td>64512/2^{25}</td>
</tr>
<tr>
<td>S3.il</td>
<td>7</td>
<td>10.52</td>
<td>19149</td>
<td>4108</td>
<td>167648/2^{27}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>7</td>
<td>9.16</td>
<td>20998</td>
<td>3285</td>
<td>174112/2^{27}</td>
</tr>
<tr>
<td>S1</td>
<td>16</td>
<td>28.54</td>
<td>30921</td>
<td>5300</td>
<td>622752/2^{28}</td>
</tr>
<tr>
<td>S2</td>
<td>16</td>
<td>62.90</td>
<td>49141</td>
<td>5568</td>
<td>1.167520/2^{29}</td>
</tr>
<tr>
<td>S3.il</td>
<td>16</td>
<td>73.21</td>
<td>41758</td>
<td>8092</td>
<td>1.567330/2^{31}</td>
</tr>
<tr>
<td>S3.ml</td>
<td>16</td>
<td>80.48</td>
<td>47646</td>
<td>5726</td>
<td>1.689950/2^{31}</td>
</tr>
</tbody>
</table>

Table 6.3. Results for the island traffic controller
In general, an arbitration circuit must fulfill the following requirements:

- At each point of time, at most one component may access the common resource:

\[ \forall t. \bigwedge_{k=0}^{n-1} \left( \text{ack}^{(t)}_k \rightarrow \bigwedge_{j=0, j \neq k}^{n-1} \neg \text{ack}^{(t)}_j \right) \]

- Only components have access to the resource, which request for it:

\[ \forall t. \bigwedge_{k=0}^{n-1} \left( \text{ack}^{(t)}_k \rightarrow \text{req}^{(t)}_k \right) \]

- The arbitration is fair, i.e. after some time each component obtains access to the resource:

\[ \bigwedge_{k=0}^{n-1} \neg \left( \forall t. \text{req}^{(t)}_k \land \neg \text{ack}^{(t)}_k \right) \]

- If none of the components wants to access the resource at a certain point of time, then at the next point of time, no persistence flag will be set and the access is given by the following static priorities:

\[ \forall t. \left( \bigwedge_{k=0}^{n-1} \neg \text{req}^{(t)}_k \right) \rightarrow \left( \bigwedge_{k=0}^{n-1} \text{ack}^{(t+1)}_k = \text{req}^{(t+1)}_k \land \bigwedge_{j=0}^{k-1} \neg \text{req}^{(t+1)}_j \right) \]

As each cell has two flags, the circuit for \( n \) components has at most \( 2^{2^n} \) reachable states. However, as always exactly one cell has the token, it can be easily seen that the circuit has only \( n2^n \) reachable states (there are \( n \)
possibilities for the flags \((\text{token}_0, \ldots, \text{token}_{n-1})\) and \(2^n\) possibilities for the flags \((\text{pers}_0, \ldots, \text{pers}_{n-1})\). Some data of the verification of the circuit for some number of components is given in the following table and also in figure 6.10. It can be easily proved that the size of the specifications is of \(O(n^2)\) and that the runtime is also polynomial [McMi93a].

![Runtime vs Components](image)

**Fig. 6.10.** Runtimes for the verification of the arbitration circuit

The automatic verification is in this case completely sufficient. Nevertheless, let us consider also a verification of the generic n-component case. The problem is here that the arbiter itself is not given as a recursive module, but as a generic one that is based on a recursive module. For this reason, we have to define a new subcircuit called `ARBIT_CHAIN` (gray shaded in figure 6.9), that can be recursively defined. `ARBIT_CHAIN` has the following specification:
ARBIT_CHAIN of length \( n + 1 \) can be defined by simply adding an arbitration cell to the chain of length \( n \). The correctness proof of ARBIT_CHAIN can be done by a simple induction proof on the number of components. The resulting proof is easily obtained by rewriting with the additional equations \( \text{token}_{\text{in}} = \text{token}_{\text{out}} \) and \( \text{gin} = \text{over}_{\text{out}} \).

6.6 Von Neumann Adder

The sequential adder of this circuit goes back to John v. Neumann. A simple implementation of the algorithm is shown in figure 6.12. The behavior of the circuit is as follows: Initially, the circuit is ready for new computations. This is indicated by the output \( \text{rdy} \). If the circuit is ready at any point of time, and a new computation is requested at \( \text{req} \), then the inputs \( A \) and \( B \) are read into the registers \( R_A \) and \( R_B \) and the circuit switches into computation mode. In this mode, the circuit performs iterations and ignores further requests. In the computation mode, the output \( \text{rdy} \) is false. If the computation terminates, then the circuit leaves the computation mode \( \text{rdy} = 1 \) and stores the results until a new computation is requested. Note that the number of required iteration steps depends on the given numbers and that an upper bound is the length \( n \) of the given bitvectors. The implementation of the controller is given in Figure 6.11.

The correctness of the above algorithm is based on the following theorem, which guarantees that the sum of \( R_A \) and \( R_B \) remains unchanged during the iterations:

\[
\left( \sum_{i=0}^{n} a_i 2^i \right) + \left( \sum_{i=0}^{n} b_i 2^i \right) = 2 \left( \sum_{i=0}^{n} (a_i \land b_i) 2^i \right) + \left( \sum_{i=0}^{n} (a_i \oplus b_i) 2^i \right)
\]

A specification of the circuit, whose verification is outlined in detail in the following, is therefore the following: \( \text{rdy} \land \text{req} \rightarrow [(C = (A \oplus B)) \land \text{rdy}] \). The
verification of this specification can be done by various strategies, however, as
many bitvector operations are involved, we suggest to verify at the bitvector
level. Figure 6.13 shows the runtimes (in seconds) and figure 6.14 shows the
storage requirements (in BDD nodes) on a Sun Sparc 10 for some strategies.

The simplest strategy that can be used is $\Phi_{F_F} \circ \Theta_B$, i.e. translating
the data equations into propositional logic and the temporal abstraction into a
prefix formula. Finally, symbolic model checking is used as a decision pro-
cedure. It is well-known that the size of the BDDs is important for the
efficiency of the verification and depends crucially on the variable order-
ing. For the verification of the adder, the following ordering has been used:
\[
\text{req} < q < a_0 < b_0 < a_1 < b_1 \ldots a_{n-1} < b_{n-1}.
\]

If the number of iteration steps did not depend on the data, it is possible
to replace the sequential circuit by a sequence of operation units, such that
a combinational circuit is obtained which can then be verified by simple
tautology checking. The number of iteration steps of the von Neumann adder
however depends on the inputs. Nevertheless, this ‘unrolling’ can also be used
for the verification of the adder, since an upper bound for the number of
iterations is known and the values of the registers do not change if further
iterations are performed after termination. The required results are given in

In order to enhance the efficiency of the verification, invariants can be used
to use the basic design ideas of the circuit. In this case, it is important to
observe that the sum of the registers remains unchanged during the iteration,
i.e. the formula $J_1 := q \land (A \oplus B) = (R_A \oplus (R_c \gg R_B))$ is an invariant of
the loop ($(R_c \gg R_B)$ means the bitvector which is obtained by concatenating $R_c$
to the left hand side of $R_B$, and $(A \oplus B)$ means a bitvector addition). This
invariant allows the elimination of all temporal operators. For reasons of
readability, define the values $R_A^o$, $R_B^o$ and $R_c^o$ of $R_A$, $R_B$ and $R_c$ at the next
point of time by $R_A^o := (R_A \oplus R_B)$, $R_B^o := \text{LSH} (R_A \uparrow R_B)$ and $R_c^o :=
R_c \lor \text{HD} ((R_A \uparrow R_B))$. Then the following subgoal remains to be proved:

![MODULE OneLoopCntr1(req, rdy1)
C1 : Not(rdy); C2 : Or(req, C1.o); C3 : Dflipflop(C2.o); C4 : And(req, rdy); C5 : Or(C3.o2, rdy1);
OUTPUT
r_\text{dy} := C5.o; ini := C4.o; load := C3.o;
END;

Fig. 6.11. Implementation of the Controller]
The number of variables can be reduced by using the more detailed invariant $J_2 := q \land \left( (A \oplus B) \sim_N (R_A \oplus (R_c \gg R_B)) \right) \land (R_c \rightarrow \Phi_N \left( (R_A \oplus R_B) < 2^n \right) )$ instead of $J_1$ (again, $(R_c \gg R_B)$ means the bitvector which is obtained by concatenating $R_c$ to the left hand side of $R_B$, $(A \oplus B)$ means a bitvector addition, and $(A \sim_N B)$ means that the bitvectors $A$ and $B$ represent the same number, i.e. differ only on leading zeros). $J_2$ contains explicitly the knowledge that there is at least one carry, i.e. if $R_c = 1$ holds, then the remaining sum of $R_A$ and $R_B$ has only $n$ bits, otherwise the sum may have $n + 1$ bits. Figure 6.13 and 6.14 show the detailed results of both invariant approaches. In this example it can be seen that the more information is given to the system, the more efficiently the circuit can be verified. However, this observation is not always valid as shown in the next section.

A generic proof has also been done for the circuit that makes use of the listed invariants. It is obvious that all used operators can be defined recursively on the length of the bitvectors. The correctness proof is then simply done by induction on the length of the arguments.
6.7 Sequential Multiplier Circuits

The circuit given in figure 6.15 implements a Russian multiplier with the same controller for the loop as in the previous section. The previously presented strategies can also be used for the verification of this circuit, the results are given in table 6.4 (runtime again in seconds; storage in number of BDD nodes).

<table>
<thead>
<tr>
<th>Bits</th>
<th>direct</th>
<th>Invariant</th>
<th>unrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Storage</td>
<td>Time</td>
</tr>
<tr>
<td>2</td>
<td>0.22</td>
<td>3254</td>
<td>0.11</td>
</tr>
<tr>
<td>3</td>
<td>0.38</td>
<td>9725</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0.99</td>
<td>10923</td>
<td>0.74</td>
</tr>
<tr>
<td>5</td>
<td>3.32</td>
<td>31024</td>
<td>3.42</td>
</tr>
<tr>
<td>6</td>
<td>15.63</td>
<td>116755</td>
<td>20.13</td>
</tr>
<tr>
<td>7</td>
<td>107.94</td>
<td>467636</td>
<td>214.38</td>
</tr>
<tr>
<td>8</td>
<td>1203.04</td>
<td>1901547</td>
<td>3416.04</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 6.4. Results for the Russian multiplier

The first column shows the results of strategy $\Phi_{PF} \circ \Theta_B$. The corresponding prefix formula of the $n$-bit circuit has exactly $(n + 2)2^{2n} + 2^n$ reachable states and can not be verified for more than 8 bits (this required 31 MByte).
Using the invariant \( q \wedge ((A \overline{\times} B) \sim \forall N (R_{C} \overline{\oplus} \text{LAST}_N 2^{n} ((R_{A} \overline{\times} R_{B})))) \), it is also only possible to verify at least 8 bits. Unrolling the circuit allows even to verify the 11 bit circuit, but more than 11 bits can not be verified with this approach. The strategies which were quite successful for the adder are not applicable for the verification of this circuit. In contrast to the adder, the complexity of this verification problem stems from the data equations and only minor complexity stems from the control part. It is however well-known that propositional formulae for multiplication have no variable ordering which gives the corresponding BDDs a non-exponential size. Hence, strategy \( \Theta_B \) is not the best choice for this circuit. Instead, the \( n \)-bit circuit has to be verified by induction for handling the data equations and invariants for handling the control part. This strategy required a lot of user interaction and cannot be explained here in detail. For example, the following lemmas had to be proved:

\[
(y \mod 2) = 0 \vdash (2 \times x) \times ((y \div 2)) = x \times y \\
(y \mod 2) = 1 \vdash x + (2 \times x) \times ((y \div 2)) = x \times y
\]

The Russian multiplier given in figure 6.15 is usually not used, since it contains more flipflops than necessary. A more efficient (in terms of chip size) sequential multiplier is given in figure 6.16. However, this multiplier is slower, since it requires always \( n \) iteration steps in contrast to the multiplier of figure 6.15 that required at most \( n \) iteration steps. The savings of chip area lead
however not to any savings for the complexity of the verification. The results for the multiplier of figure 6.16 are given in table 6.5.

6.8 Systolic Arrays

6.8.1 A Simple Version. This circuit has the task to compute the scalar product of a sequence of inputs in a sequential manner, i.e. the components of the vectors are given sequentially. More formally, the circuit has three m-bit inputs $A_{in}$, $B_{in}$ and $C_{in}$ and one m-bit output $C_{out}$. The specification of the circuit is as follows:

$$\forall t. C_{out}^{t+n} = C_{in}^{t} + \sum_{i=1}^{n} A_{in}^{t} \times B_{in}^{t}$$

The circuit is parameterized in two ways: first there is a parameter $n$ which is the length of the input vectors and second there is a parameter $m$ which is the bitwidth of the components of the vectors. The implementation of the circuit is given by a sequence of base cells, i.e. a one dimensional systolic array (figure 6.17). The inputs $A_{j}$ and $B_{j}$ of cell $j$ are just passed to the outputs $A_{j+1}$ and $B_{j+1}$ of the cell, while the output $C_{n-j}$ of cell $j$ is the sum of the input $C_{n-j}$ and the product of $A_{j}$ and $B_{j}$ of the previous point of time.
The behavior of a single cell determines the relations between the signals and we obtain the following description of the signal flow for $j \in \{0, \ldots, n - 1\}$:

\begin{align*}
(1) \quad A_{t+1}^j & := A_t^j \\
(2) \quad A_0^{j+1} & := 0 \\
(3) \quad A_0^t & := A_{in}^t \\
(4) \quad B_{t+1}^j & := B_t^j \\
(5) \quad B_0^{j+1} & := 0 \\
(6) \quad B_0^t & := B_{in}^t \\
(7) \quad C_{t+1}^j & := C_t^j + A_{n-(j+1)}^t \times B_{n-(j+1)}^t \\
(8) \quad C_0^{j+1} & := 0 \\
(9) \quad C_0^t & := C_{in}^t
\end{align*}

Equation (7) is not given immediately from the implementation. From figure 6.17, it can be derived that $C_{t+1}^{n-j+1} = C_{n-j}^t + A_j^t \times B_j^t$ holds for $j \in \{1, \ldots, n\}$. If we substitute $j := n - i$, we obtain: $C_{t+1}^{n-(n-i)+1} = C_{n-(n-i)}^t + A_{n-i}^t \times B_{n-i}^t$
which is obviously equation (7). To show the correctness of the circuit, we first prove the following conditions for the signals $A_j$ and $B_j$:

\[(10) \forall j \leq n. \forall t. A_j^{t+j} = A_{in}^t \quad (11) \forall j \leq n. \forall t. B_j^{t+j} = B_{in}^t\]

Both propositions can be proved via induction on $j$. In the base case, we have $\forall t. A_0^{t+0} = A_{in}^t$, which does hold according to equation (3). In the induction step, we have $A_{j+1}^{t+j+1} = A_j^{t+j}$ (10). The proof of (11) is done analogously. Now, we can prove

\[(12) \forall j \leq n. \forall t. C_j^{t+j} = C_{in}^t + \sum_{i=1}^{j} A_{n-i}^{t+i-1} \times B_{n-i}^{t+i-1}\]

Again, we perform induction on $j$: the base case is trivial, since we have $\forall t. C_0^{t+0} = C_{in}^t + \sum_{i=1}^{0} = C_{in}^t$ which holds by equation (9). The induction step is proved as follows:

\[
C_{j+1}^{t+j+1} = C_j^{t+j} + A_{n-(j+1)}^{t+j} \times B_{n-(j+1)}^{t+j}
\]

\[
= C_{in}^t + \left( \sum_{i=0}^{j} A_{n-i}^{t+i-1} \times B_{n-i}^{t+i-1} \right) + A_{n-(j+1)}^{t+j} \times B_{n-(j+1)}^{t+j}
\]

Our final aim is to prove the flow of the output $C_{out}$. The combination of the equations (10), (11) and (12) yields in:

\[(13) \forall t. C_{out}^{t+2n} = C_{in}^{t+n} + \sum_{i=1}^{n} A_{in}^{t+2i-1} \times B_{in}^{t+2i-1}\]

In order to do the proofs by RRL, the equations (1-9) are introduced by the translation of the circuit structure into equation systems. Furthermore, the following equations are added by the translation of the abstract data type definition with its recursive definition of the operators below):
Equations (14) and (15) are the axioms for ≤. Equation (18) is not necessary, but leads to shorter terms in that the product of $A_i^t$ and $B_i^t$ is abbreviated by $M_i^t$. Finally, equations (19) and (20) define the scalar product of $A_i^t$ and $B_i^t$. Note that the scalar product is normally a function in the input vector $A_i^t$ and $B_i^t$. Such a view on the scalar product makes it however a higher-order function that can not be handled by first order term rewriting. Hence, we define the scalar product only for our given vectors. Obviously, we have $S_{j,n} = \sum_{i=1}^j M_{n-i}^{t+i-1} = \sum_{i=1}^n A_{n-i}^{t+i-1} \times B_{n-i}^{t+i-1}$.

6.8.2 A more complicated version. This array is essentially the array as given in the last section. However, there is no input $B$ anymore. Instead the scalar product is done by fixed weights that are stored in the cells. In an initial phase, these weights are loaded from the input $A$. From figure 6.18, we can read the following equations for $j \in \{0, \ldots, n - 1\}$:

- $A_{j+1}^{t+1} := A_j^t$ (1)
- $A_{j+1}^0 := 0$ (2)
- $A_0^t := A_{in}$ (3)
- $B_{j+1}^{t+1} := B_j^t + A_{n-(j+1)}^t \times W_{n-j}^t$ (4)
- $B_{j+1}^0 := 0$ (5)
- $B_0^t := B_{in}$ (6)
- $W_{j+1}^{t+1} := (\text{reset}(t) \Rightarrow A_j^t \mid W_{j+1}^t)$ (7)
- $W_0^{t+1} := 0$ (8)
316 Klaus Schneider and Thomas Kropf

$W_j^t$ is the value of the weight that is stored by cell $j$ at time $t$. We want to prove a relationship between the signals $A_{in}^t$ and $B_{out}$. For this reason, as in the last example we establish some lemmata which are helpful for this purpose. Furthermore, we give an input restriction for the input $r$: we assume that

\begin{align*}
(9) \text{reset}^{(n)} &:= 1 & (10) \forall t. \text{reset}^{(t+n+1)} &:= 0
\end{align*}

holds. This means we initialize the weights of the array at time $n - 1$ with the weights $A_{in}^0, \ldots, A_{in}^n$. In order to prove our result, we prove:

\begin{align*}
(11) \forall j \leq n. \forall t. A_{in}^{t+j} &= A_{in}^t \quad (12) \forall j \leq n. \forall t. W_{j+1}^{t+n+1} = A_j^n
\end{align*}

(11) can be proved via induction on $j$. In the base case, we have $\forall t. A_{0}^{t+0} = A_{in}^0$, which does hold according to equation (3). In the induction step, we have $A_{j+1}^{t+j+1} = A_j^{t+j}$. (12) is proved by induction on $t$: in the base case, we have $W_{j+1}^{n+1} = 0$, and in the induction step $W_{j+1}^{t+n+2} = W_{j+1}^{t+n+1} + A_{j+1}^{t+n+1}$.

We know that the array has been initialized correctly, i.e. at time $n$ all weights are initialised such that (12) holds.

Now we can draw our attention to the streams $B_j$ and prove

\begin{align*}
(13) \forall j \leq n. \forall t. B_{j}^{t+j} &= B_{in}^t + \sum_{k=0}^{j-1} A_{n-(k+1)}^{t+k} \times W_{n-k}^{t+k}
\end{align*}

This is proved by induction on $j$. The base case holds, since $B_0^{t+0} = B_{in}^t = B_{in}^t + \sum_{k=0}^{j-1} A_{n-(k+1)}^{t+k} \times W_{n-k}^{t+k}$. The induction steps is proved as follows:

\begin{align*}
B_{j+1}^{t+j+1} &= B_{j}^{t+j} + A_{n-(j+1)}^{t+j} \times W_{n-j}^{t+j} \\
&= B_{in}^t + \left( \sum_{k=0}^{j-1} A_{n-(k+1)}^{t+k} \times W_{n-k}^{t+k} \right) + A_{n-(j+1)}^{t+j} \times W_{n-j}^{t+j} \\
&= B_{in}^t + \left( \sum_{k=0}^{j} A_{n-(k+1)}^{t+k} \times W_{n-k}^{t+k} \right)
\end{align*}

In particular ($j := n$), we find after a simple index transformation:

\begin{align*}
(13a) \forall t. B_{out}^{t+n} &= B_{in}^t + \sum_{j=1}^{n-1} A_{j-1}^{t+n-j} \times W_{j}^{t+n-j}
\end{align*}

If we instantiate $t+n$ for $t$ in equation (13a) and use equation (11), we obtain finally the following behavior of the circuit:

\begin{align*}
(13b) \forall t. B_{out}^{t+2n} &= B_{in}^{t+n} + \sum_{j=1}^{n-1} A_{in}^{t+2(n-j)+1} \times W_{j}^{t+2n-j}
\end{align*}
Using again an index transformation this is the same as

\[
(13c) \forall t. B_{out}^{t+2n} = B_{in}^{t+n} + \sum_{k=1}^{n-1} A_{in}^{t+2k+1} \times W_j^{t+n+k}
\]

Hence, we see that the circuit computes two scalar products, one at the even points of time and another one at the odd points of time.

6.9 GSM Speech Encoding

6.9.1 The Algorithm. The verification example given in this section is part of the European digital cellular telecommunication system, in particular it belongs to a 'full rate speech transcoder'. It's implementation is based on the standardisation document [GSM94] of the European Institute for Telecommunication Norms.

A detailed evaluation of the entire description of [GSM94] shows that in the context of a HW/SW codesign the hardware implementation of the module for the short term analysis filtering (STAF for short) leads to a significant enhancement of the performance of the full rate speech transcoder. In this section, a hardware implementation generated by a high-level synthesis tool and the verification of the module is shown.

The short term analysis filtering due to [GSM94] requires special arithmetic functions `add` and `mult_r` which are defined in the standard document [GSM94] for signed integers. Bitvectors are thereby interpreted as signed integers by the following function \( \Phi_s \) that interprets the bitvectors as two-complement encodings:

\[
\Phi_s([b_n, \ldots, b_0]) := -b_n \cdot 2^n + \sum_{i=0}^{n-1} b_i \cdot 2^i
\]

Given a bitwidth \( n + 1 \), i.e. bitvectors of the form \([a_n, \ldots, a_0]\), the largest representable number is \( \text{Max}_{\text{Int}} := 2^n - 1 \) and the smallest (negative) number \( \text{Min}_{\text{Int}} := -2^n \). The definition of the functions `add` and `mult_r` used in the definition of STAF are defined as follows: `add(x, y)` performs the addition of \( x \) and \( y \) with overflow control and saturation; the result is set at \( \text{Max}_{\text{Int}} \) when overflow occurs or at \( \text{Min}_{\text{Int}} \) when underflow occurs. `mult_r(x, y)` performs multiplications of \( x \) by \( y \) and scales the result (with rounding). The formal definition of the two operations in terms for the bitwidth \( n + 1 \) is as follows:

`add(x, y) := \begin{cases} 2^n - 1 & : \text{overflow}(x + y) \\ -2^n & : \text{underflow}(x + y) \\ x + y & : \text{otherwise} \end{cases} = \begin{cases} \text{Max}_{\text{Int}} & : \text{overflow}(x + y) \\ \text{Min}_{\text{Int}} & : \text{underflow}(x + y) \\ x + y & : \text{otherwise} \end{cases}`

`mult_r(x, y) := \begin{cases} 2^n - 1 & : x = y = -2^n \\ (x \cdot y + 2^n - 1) \gg n & : \text{otherwise} \\ \text{Max}_{\text{Int}} & : x = y = \text{Min}_{\text{Int}} \\ (x \cdot y + 2^n - 1 \text{ DIV } 2^n) & : \text{otherwise} \end{cases}`
Division of $x$ by $2^m$ means in the bit representation of the number $x$ a right shift by $m$ bits, where the sign bit (the most significant and also the leftmost one) is replicated. Given two signed $n + 1$-bit numbers $x$ and $y$ in 2-complement, the bitwidths of $\text{add}(x, y)$ is also $n + 1$, since the cases where either an underflow or an overflow occurred are defined as $\text{Min}_\text{Int}$ and $\text{Max}_\text{Int}$, respectively. $\text{mult}_r(x, y)$ also requires only $n + 1$ bits since $2n - 1$ bits are sufficient to store the intermediate result $(x \cdot y) + 2^{n-1}$:

$$\text{mult}_r(\Phi_s(A), \Phi_s(B)) \leq (-2^n)(-2^{n-1}) + 2^{n-1} \text{ DIV } 2^n$$

$$= [2^n - 1 + \frac{1}{2}] = 2^n - 1 = \text{Max}_\text{Int}$$

$$\text{mult}_r(\Phi_s(A), \Phi_s(B)) \geq ((-2^n)(2^{n-1} + 2^{n-1} \text{ DIV } 2^n)$$

$$= [-(2^n - 1) + \frac{1}{2}] = -(2^n - 1) > \text{Min}_\text{Int}$$

![Fig. 6.19. Implementation of the GSM addition](image1)

![Fig. 6.20. Implementation of the GSM multiplication](image2)
Note that \(\lfloor x + \frac{1}{2} \rfloor = x\) for \(x \in \mathbb{Z}\). Combinational hardware implementations for the computation of these arithmetic operations are given in figures 6.19 and 6.20. Note that in figure 6.20 the result of the adder has \(2n + 2\) bits, i.e. the array \(S[0 \ldots 2n + 1]\), and that the leftmost bit as well as the lowest \(n - 1\) bits are neglected. This is legal due to the inequalities above that assert that the result can be stored with \(n + 1\) bits. Note that the \(\text{mult}_r(\text{Min}_\text{Int}, \text{Min}_\text{Int})\) has to be declared in a special case as the usual term would lead to an overflow, as can be seen as follows:

\[
\left((-2^n)(-2^n) + 2^{n-1}\right) \div 2^n = \left(2^{n+1} + 1\right) \div 2^n = 2^n = \text{Max}_\text{Int} + 1
\]

The implementation of \(\text{mult}_r\) as given in figure 6.20 has however the disadvantage that intermediate computation steps have a bitwidth larger than \(n + 1\). This has serious drawbacks when the multiplication, the addition of \(2^{n-1}\) and the shift-operation are done at different points of time, such that the intermediate results have to be stored. The implementation given in figure 6.21 removes this drawback. The implementations given in figures 6.20 and 6.21 can be shown to be equivalent to each other. The correctness can be shown as follows:

\[
\left(\Phi_s(A) \cdot \Phi_s(b) + 2^{n-1}\right) \div 2^n = \left(\left(\Phi_s(A) \cdot \Phi_s(b) \div 2^n\right) + 1\right) \div 2^n
\]

\[
\begin{cases}
(\Phi_s(A) \cdot \Phi_s(b) \div 2^n) : \text{for } \Phi_s(A) \cdot \Phi_s(b)[n - 1] = 0 \\
(\Phi_s(A) \cdot \Phi_s(b) \div 2^n) + 1 : \text{for } \Phi_s(A) \cdot \Phi_s(b)[n - 1] = 1
\end{cases}
\]

Fig. 6.21. Implementation of the GSM multiplication

A description of the short term analysis filtering algorithm of section 4.2.10 in [GSM94] in the programming language C is given in figure 6.22. The algorithm needs to keep the array \(u[0 \ldots 7]\) in memory for each call, and the initialisation of this array is 0.
static void STAF(int u[8], rp[8], k, s[160])
{
    int d, z, ui, sav, rpi, i, j;
    for(k=0; k < k; k++)
    {
        d = s[k];
        sav = d;
        for(i=0; i<8; i++)
        {
            ui = u[i];
            rpi = rp[i];
            ui = sav;
            z = mult_r(rpi, d);
            sav = add(ui, z);
            z = mult_r(rpi, ui);
            d = add(d, z);
        }
        s[k] = d;
    }
}

Fig. 6.22. The short term analysis filtering algorithm in C

Note that the given function STAF overwrites the array s with the computed values. The standard document does not allow this, instead a new array d[0...159] is to be assigned. However, the given C function does only model the hardware module and before it is invoked, the the array s is copied. After termination of the function, the values of the array d can be read from the local array s. An unrolling of the loops body leads to the following recursive equations that are also given in section 3.1.11 of [GSM94]:

\[
\begin{align*}
    d_{0,k} &:= s_k \\
    u_{0,k} &:= s_k \\
    d_{i,k} &:= d_{i-1,k} + r_{p_i} \cdot u_{i-1,k-1} & i &\in \{1, \ldots, 8\} \\
    u_{i,k} &:= u_{i-1,k-1} + r_{p_i} \cdot d_{i-1,k} & i &\in \{1, \ldots, 8\} \\
    d_k &:= d_{8,k}
\end{align*}
\]

\(d_{i,k}\) is the value of the output \(d\) of the \(i-1\)-th row in the \(k\)-th column (for \(i \in \{1, \ldots, 8\}\)) and \(d_{0,k} = s_k\). \(u_{i,k}\) is the value of the output \(u_i\) of the \(i\)-th row in the \(k\)-th column (for \(i \in \{0, \ldots, 7\}\)). It is easy to see that the following equations hold:

\[
\begin{align*}
    d_{i,k} &= s_k + \sum_{j=1}^{i} r_{p_j} u_{j-1,k} \\
    u_{i,k} &= s_k + \sum_{j=1}^{i} r_{p_j} d_{j-1,k}
\end{align*}
\]

It is possible to implement the entire function as a combinatorial circuit according to the above equations. Figure 6.23 shows an example implementation where \(8 \times k\) modules are required that are themselves implemented by
two GSM multipliers and two GSM adders (see figures 6.19 and 6.20). The implementation of figure 6.23 is a matrix with 8 columns and \( k_n \) rows. The initial values \( u[i] \) in row \( k = 0 \) are 0 according to the standard document.

![Diagram of STAF implementation](image)

**Fig. 6.23.** Implementation of STAF as combinatorial circuit.

Figure 6.23 should be read as rows of columns, since each row corresponds to the execution of the inner loop. The figure illustrates the data flow through the computation in the module. While the array \( r_p \) is not changed, the array \( u \) is updated every time the inner loop is performed. After termination of the inner loop, the value \( d[k] \) is available (and stored in \( s[k] \) in the C version).

For obvious reasons, some parts of the circuit have to be implemented in a sequential manner such that the number of GSM multipliers and GSM adders can be reduced, i.e. the same GSM multipliers and GSM adders can be used for different computations. This requires the implementation of a controller that manages the different computations in the data path. In the next section, a sequential implementation of STAF is presented that requires only one (GSM) multiplier and one (GSM) adder.

### 6.9.2 Register-Transfer Implementation of STAF

A sequential hardware implementation of STAF must first schedule the arithmetical operations to certain points of time. A simple way to do this without further optimizations is to map each instruction of the C code to one clock cycle. A result of such a simple mapping leads then to the algorithm of figure 6.24, where some statements are labeled with corresponding control states \( S_0, \ldots \).

The next step is to implement the control flow of the program and the corresponding data path. The sequential implementation of STAF based on figure 6.24 contains two data paths and a controller. The first data path
\[ k := 0 \]
\[
\begin{align*}
S0 : & \quad R6 := 0 \text{ falls Start} \\
S3 : & \quad l_8 := R6 < k_n \\
k < k_n? & \quad \text{IF } l_8 \text{ THEN } S4 \text{ ELSE } S0; \\
d = s[k]; & \quad R4 := s_{\text{read}} = \text{Store}[R6 + s] \\
i := 0; & \quad R7 := 0 \\
sav := d; & \quad R5 := R4 \\
k < k_n? & \quad l_8 := R7 < 8; \\
iu := u[i]; & \quad \text{IF } l_8 \text{ THEN } S8 \text{ ELSE } S37; \\
rpi := rp[i]; & \quad R2 := r_{\text{read}} = \text{Store}[R7 + rp] \\
\text{s}[i] := \text{sav} & \quad \text{IF } M_R \text{ THEN } R3 := \text{Max}_\text{Int} \\
z := \text{mult}_r(rpi, d) & \quad \text{IF } K_R \text{ THEN } R3 := 1 + \text{Max}_\text{Int} \\
\text{sav} := \text{add}_r(ui, z) & \quad R3 := 1 + R1 + R3 \quad \text{inflow} := \text{Min}_\text{Int} < R3 \lor R3 > \text{Max}_\text{Int} \\
z := \text{mult}_r(rpi, ui) & \quad g_{32} := \text{MSB}(R3) \land \text{inflow}; \quad \text{overflow} \\
d := \text{add}_r(d, z) & \quad \text{if } g_{32} \text{ then } S29 \text{ else } S23; \\
i := i + 1; & \quad \text{if } g_{32} \text{ then } S28 \text{ else } S26; \\
s[k] := d & \quad \text{if } g_{32} \text{ then } S31 \text{ else } S29; \\
k := k + 1 & \quad \text{if } g_{32} \text{ then } S33 \text{ else } S31; \\
\end{align*}
\]

\textbf{Fig. 6.24.} A finite state algorithm for STAF with \( n + 1 \) bits in the data path.
is used to perform the computations according to the last section, and the second data path is used for updating the loop variables $k$ and $i$.

Fig. 6.25. State transition diagram of the correct version of the STAF controller

The state transition diagram of the controller is given in figure 6.25. There is also a transition from each of the states to the initial state $S0$ that is performed when a reset signal occurs. For reasons of readability, these transitions are not drawn in figure 6.25. The output equations of the controller are given as follows:

1. $\text{Ready} := \neg \text{Reset} \land S3 \land \neg l8$
The mapping of the algorithm to the circuit structure involves the following
register mapping: R1 ~ w1, R2 ~ rpi, R3 ~ z, R4 ~ d, R5 ~ sav, R6 ~ k,
and R7 ~ i. The data paths of the circuit are given in figures 6.26 and 6.27,
where a multiplex-based implementation has been chosen. The data path
of figure 6.27 is used for controlling the loops and the data path of figure 6.26
performs the additions and multiplications for the filtering function.

The standard requires that the bitwidths of the upper half are 16 bits,
but the verification can also be done with smaller bitwidths, if the constants
Max.Int and Min.Int are adapted properly in the data path of figure 6.26. In
contrast to the data path of figure 6.26, the values in the registers R6 and R7
can be interpreted as unsigned integers in figure 6.27. It has to be noted, that
register R7 is used for storing the loop variable i and hence, only contains
the values 0, ..., 7. Bitwidths wider than 3 bits for R7 do not make sense
at all. Register R6 corresponds to the loop variable k and runs through the
values $0, \ldots, k_n$. An upper bound for $k_n$ is 159, and hence a bitwidth greater than 8 for $R7$ does also not make sense.

6.9.3 Specifications and Verification Results. In order to verify the correctness of the circuit, the circuits 6.19 and 6.20 are used to implement verification circuits according to figures 6.29 and 6.29.

The circuit given in figure 6.29 is used to check the correctness of the loop control of both loops. There are two states of the controller that are important for the verification of the control behaviour: the initial state $S0$ and the state $S7$, where it is checked whether the inner loops is to be terminated or not. In
Fig. 6.27. Data path of the STAF module for controlling loop variables.

Fig. 6.28. Verification of the loop control

particular the following specifications have to be checked (note that the inner loop variable \(i\) is mapped to \(R7\) and that the outer loop variable is mapped to \(R6\)):

**Initial Control State.** Initially, the controller is in state \(S0\) and the next state is also \(S0\) when a reset occurs:

C1a: \(S0\)
C1b: \(G[\text{reset} \rightarrow X\cdot S0]\)

**Initialisation of the Loop Variables.** If a start signal occurs, then the controller moves to state \(S7\) and sets the registers \(R6 = R7 = 0\):

C2a: \(G[\neg \text{reset}] \rightarrow G[S0 \land start \rightarrow X[R6 = 0]]\)
C2b: \(G[S0 \rightarrow (((R6 = 0) \land (R7 = 0)) \lor S7)]\)
C2c: \(G[\neg \text{reset}] \rightarrow G[S0 \land start \land X[k_n \neq 0] \rightarrow X[XX[XX[S7 \land (R6 = 0) \land (R7 = 0)]]]]\)
Updating Loop Variables during inner Loop. If the inner loop's body is executed then the inner loop variable i (R7) is incremented (C3a) and the outer loop variable k (R6) is not changed (C3b):

C3a: \( G(\neg \text{reset}) \rightarrow G[S7 \land (R7 < 8) \rightarrow \chi ((R7 = D) \wedge S7)] \)
C3b: \( G(\neg \text{reset}) \rightarrow G[S7 \land (R7 < 8) \rightarrow [(R6 = XR6) \lor S7]] \)

Terminating the inner loop. If the outer loop's body is executed (after inner loop has been executed) and another iteration of the outer loop is necessary, i.e. \( k < k_n \) \((R6 + 1 < k_n \text{ or } C + 1 < k_n)\), then the inner loop variable i (R7) is reinitialized to 0 (C4a) and the outer loop variable k (R6) is incremented (C4b):

C4a: \( G(\neg \text{reset}) \rightarrow G[S7 \land \neg(R7 < 8) \rightarrow X[\neg (R7 = 0) \wedge S7]] \)
C4b: \( G(\neg \text{reset}) \rightarrow G[S7 \land \neg(R7 < 8) \land X(C < k_n) \rightarrow [(\neg S0) \lor S7]] \)
C4c: \( G(\neg \text{reset}) \rightarrow G[S7 \land \neg(R7 < 8) \land X(C < k_n) \rightarrow \chi ((a6 = C \land R7 = 0) \wedge S7)] \)

Terminating the outer Loop. If the outer loop's body is executed (after inner loop has been executed) and another no further iteration of the outer loop is necessary, then the controller moves to the initial state S0:

C5: \( G(\neg \text{reset}) \rightarrow G[S7 \land \neg(R7 < 8) \land \neg X(C < k_n) \rightarrow XXXXS0] \)

The above specifications describe the entire behaviour of the loop control. Therefore, only the states S0 and S7 have to be considered. The runtimes for the verification of the above specifications with the LTL model checker of C@S are given in table 6.6.

<table>
<thead>
<tr>
<th>Spec</th>
<th>Runtime [sec]</th>
<th>BDD nodes total</th>
<th>BDD nodes for Trans. Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1a</td>
<td>22.74</td>
<td>10091</td>
<td>393</td>
</tr>
<tr>
<td>C1b</td>
<td>23.11</td>
<td>10011</td>
<td>386</td>
</tr>
<tr>
<td>C2a</td>
<td>32.11</td>
<td>10362</td>
<td>421</td>
</tr>
<tr>
<td>C2b</td>
<td>24.45</td>
<td>10028</td>
<td>427</td>
</tr>
<tr>
<td>C2c</td>
<td>31.69</td>
<td>10092</td>
<td>800</td>
</tr>
<tr>
<td>C3a</td>
<td>46.94</td>
<td>10692</td>
<td>1174</td>
</tr>
<tr>
<td>C3b</td>
<td>48.68</td>
<td>10469</td>
<td>1397</td>
</tr>
<tr>
<td>C4a</td>
<td>45.58</td>
<td>10472</td>
<td>1153</td>
</tr>
<tr>
<td>C4b</td>
<td>60.56</td>
<td>10664</td>
<td>1328</td>
</tr>
<tr>
<td>C4c</td>
<td>58.36</td>
<td>11584</td>
<td>1378</td>
</tr>
<tr>
<td>C5</td>
<td>87.59</td>
<td>22563</td>
<td>3731</td>
</tr>
</tbody>
</table>

Table 6.6. Runtimes for the STAF control behavior.

The verification of the data path that computes the filtering (figure 6.26), similar hardware extensions have been added to the circuit (see figure 6.29). The circuit given in figure 6.29 is used to check the correctness of the filter function. The additional hardware contains the module that has been used in section 6.9.2 to implement a combinational circuit for STAF. It is activated,
i.e. the results of the computation is stored into the registers $A$ and $B$ exactly when the controller is in state $S9$ as this is the first state, where the inputs for the computations have been read and are available in the registers $R1$, $R2$, and $R4$. The specifications that have been checked are that the values $A := \text{add}(\text{mult}_r(rpi, ui)), d)$ and $B := \text{add}(\text{mult}_r(rpi, d)), ui)$ are computed correctly. Note that $rpi$, $ui$, and $d$ are mapped to the registers $R2$, $R1$, $R4$, respectively. The values $A$ and $B$ can be computed in state $S9$, as this is the first state where the current values of $R1$ and $R2$, i.e. $rpi$ and $ui$ are available. They are compared to the values of the registers $R4$ and $R5$ when the controller is again in state $S7$:

$$D1 : G[S9 \rightarrow [(A = R4) W S7]]$$
$$D2 : G[S9 \rightarrow [(B = R5) W S7]]$$
$$D3 : G[S9 \rightarrow [(C = R7) W S7]]$$

The runtimes for the verification of the above specifications with the LTL model checker of C0S are given in table 6.7.

<table>
<thead>
<tr>
<th>Spec</th>
<th>Runtime [sec]</th>
<th>BDD nodes total</th>
<th>BDD nodes for Trans. Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>282.55</td>
<td>327467</td>
<td>93321</td>
</tr>
<tr>
<td>D2</td>
<td>197.04</td>
<td>289418</td>
<td>74329</td>
</tr>
</tbody>
</table>

Table 6.7. Runtimes for the STAF data path.

7. Conclusions

We have presented a new approach for the verification of digital circuits. Different decision procedures are coupled by transforming specification and implementation descriptions into a common representation based on a restricted
class of higher order logic. Dependent on the type of the verification problem, the most suited solution is selected and the necessary decision procedures are invoked. In contrast to previous approaches, several solution strategies may be combined to verify circuits for which a specification includes complex timing as well as abstract data types. Using different examples, different verification strategies have been demonstrated and compared with regard to the circuit size and necessary amount of interaction. The C@S system is a research prototype and currently extended by other decision procedures and verification strategies, which have not been presented in this book section. For an actual and up-to-date overview of the current implementation status and availability see the online documentation at http://goethe.ira.uka.de/hvg/cats/.