Compilation of Imperative Synchronous Programs with Refined Clocks

Mike Gemünde, Jens Brandt and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract

To overcome over-synchronization in synchronous programs, we recently introduced clock refinement to our synchronous programming language Quartz. This extension basically allows programmers to refine reaction steps into smaller internal computation steps while maintaining the external behavior. In this paper, we consider the compilation of the extended Quartz programs to synchronous guarded actions. To this end, we first define an intermediate language supporting multiple clocks based on synchronous guarded actions which is the target of the front-end of the compiler and the source of back-end tools that perform efficient analysis and synthesis procedures. We moreover present a compilation scheme to translate the extended Quartz programs to the new intermediate language. We discuss important design considerations and illustrate our approach with the help of some small examples.

1. Introduction

A major advantage of the synchronous model of computation [1, 4, 6, 17, 24] is its temporal abstraction which divides the execution of a program into a sequence of so-called macro steps. Macro steps model the progress of time and subsume a finite number of instantaneous micro steps which represent the real computation and communication. This abstraction and the concise semantics of synchronous languages is the key (1) to precisely model synchronous hardware circuits [2, 21, 24], (2) to generate deterministic single-threaded software from multi-threaded programs [3, 12], and (3) to formally reason about programs and related tools [5, 22, 23, 25]. However, the synchronous model of computation also has some drawbacks: First, it challenges compilers: for imperative languages, a lot of problems had to be solved to deal with causality and schizophrenia problems [5, 25]. Second, it imposes some limitations to programmers concerning the temporal relationship of threads: even though some threads may not be related at all, they are nevertheless forced to run in lockstep due to the unique basic clock of the synchronous program that defines the macro steps of the threads. This phenomenon is sometimes called over-synchronization and makes refinement and reuse of existing programs difficult.

In particular, compilers are challenged when generating efficient multithreaded code for programs consisting of such loosely coupled threads, since all threads of these programs implicitly synchronize after each macro step, even if there are no data dependencies. While a data flow analysis may be able to detect the independence of threads and is then able to desynchronize such programs [7, 9], adding an explicit notion of independence allows compilers to create desynchronized code without expensive analyses since unnecessary synchronization is avoided by construction. While this is no problem when generating code containing a single thread of execution, it becomes problematic for multithreaded code generation.

In addition to the problem to generate efficient multithreaded code of synchronous programs with loosely coupled threads, also the comfort for programmers can be increased by the introduction of subclocks. Without subclocks, programmers are limited by the single temporal abstraction layer of the synchronous model of computation offered by languages like Esterel and Quartz. For example, it is not easily possible to refine a macro step of one thread into a data-dependent loop without destroying the temporal relationship to the other threads. To allow such a refinement, different levels of logical time would be required. A similar problem is obtained when modules at different abstraction levels should be combined to a single system.

In [14, 15], we therefore presented clock refinement for imperative synchronous programming languages which allows programmers to refine reaction steps into internal smaller computation steps while keeping the external deterministic behavior. Compared to Lustre [16, 17], our subclocks are used to refine the basic clock of the synchronous problem into faster and possibly independent subclocks, while Lustre follows the opposite approach where clock abstraction is used to generate slower clocks from the basic clock. Compared to the polychronous approach of Signal [13, 19, 20], our clock refinement is more restrictive, since clocks in Signal can be arbitrarily related and therefore clock consistency must be explicitly checked by means of a clock calculus.
In this paper, we consider the compilation of our imperative synchronous programs with refinement by subclocks. The contribution of this paper is thereby twofold: first, we define a new intermediate language based on synchronous guarded actions. This intermediate language resolves the complex interaction of different statements at the source code level, so that back-end tools can better concentrate on efficient analysis and code generation procedures. Second, we present a preliminary compilation algorithm that translates the extended synchronous programs to the intermediate language. Thereby, we focus on the compilation of complex control flow statements (such as different kinds of preemption) in the context of different clock levels and do not elaborate on special issues such as schizophrenic local variables or modular compilation.

The rest of the paper is organized as follows: Section 2 introduces the synchronous model of computation, sketches our clock refinement extension, and compares it with other multi-clock approaches. Section 3 presents the intermediate format together with the extensions to handle refined clocks. The compilation of synchronous Quartz programs with subclocks is presented in Section 4. A working example to explain the compilation is shown and explained in Section 5. Finally, Section 6 draws some preliminary conclusions.

2. Clock Refinement

Imperative synchronous languages, like Quartz [24] or Esterel [4] usually contain statements which divide the execution of the program into macro steps (also called reactions). Following the synchronous model of computation, these programs only exchange inputs and outputs with the environment once in such a macro step, and also all local variables have a unique value within such a macro step. Figure 1 (a) illustrates this behavior. The micro steps which represent the computation and communication within a reaction are instantaneous which means that they are executed within the same variable environment. However, to ensure causality they must be executed in a causal ordering that guarantees that there are no reads of local and output variables before the values of these variables are determined by other micro steps of the macro steps.

We recently extended our synchronous language Quartz [24] by subclocks to achieve clock refinement [14, 15]. The idea of a refined clock is to allow the programmer to divide a macro step into smaller substeps. While input and output variables still refer to the basic clock that determines the macro steps, local variables may now refer to faster subclocks. Hence, local variables may change their values several times within a macro step depending on the number of ticks of their clock within the macro step. Since only local variables refer to subclocks, the effects of clock refinement are not visible outside the module. This refinement is illustrated in Figure 1 (b) where the basic clock \( C_0 \) is refined to a clock \( C_1 \) which itself is refined to another clock \( C_2 \).

As already stated in the introduction, subclocks in our context are different to abstracted clocks as provided by Lustre. While Lustre allows developers to derive slower clocks by its down-sampling operator \( \downarrow \), the presented clock refinement creates faster clocks which is just the opposite direction. Figure 2 (a) illustrates a sample clock tree, where the marked branch shows the relation of the clocks of Figure 1 (b). The relation to clocks in Lustre is illustrated by Figure 2 (b), where the slowest clocks are located at the top and the fastest clocks are located at the bottom of the hourglass.

In our synchronous language Quartz, the extension consists of two statements. First, the statement \( \text{clock}(C) \ S \) declares a new clock \( C \) that refines the current clock so that all local variables declared in the statement \( S \) refer to this new clock \( C \). Hence, local variables declared within \( S \) can change their values in each clock tick of \( C \). Since their scope is always smaller than the scope of their corresponding clock \( C \), updates of variables bound to subclocks are not visible to the environment. As a consequence, subclocks are not visible beyond module boundaries.
Second, the statement \texttt{pause}(C)\) denotes a tick of clock \(C\) as a refinement of the already available statement \texttt{pause} which still denotes a tick of the basic clock. Since the basic clock is not explicitly declared, it is not given an explicit name, but we refer to it with \(C_0\) in the remainder of this paper.

For example, consider the program shown on the left-hand side of Figure 3. The module takes two inputs \(a\) and \(b\) and computes an output \(gcd\) which is the greatest common divisor of the both inputs. Inside the module, a subclock \(C_1\) is declared together with two local variables \(x\) and \(y\) which are bound to this clock since they are declared in the scope of \(C_1\). Then, an iteration computes the greatest common divisor and the loop makes a step on the subclock each iteration as long as the result is not obtained. When the computation terminates, the output \(gcd\) is set and also the module terminates. For the whole computation, the module needs several steps with respect to clock \(C_1\), but no step at the level of the basic clock \(C_0\). For the environment, the execution therefore seems to be instantaneous\(^1\).

The second example is given on the right-hand side in Figure 3. This program makes use of independent subclocks \(C_1\) and \(C_2\) that are declared in different threads whose execution is therefore only restricted by potential data dependencies. Thus, both threads do not need to synchronize at \(\ell_{11}, \ell_{21}\) or \(\ell_{22}\), but only at the ticks of the basic clock when locations \(\ell_{1}\) and \(\ell_{2}\) are reached. The variables \(x, y\) and \(z\) are all declared at the highest clock level which is the basic clock visible at the interface of the module. In contrast, the local variable \(a\) is declared in the scope of \(C_1\) and thereby can change with ticks of \(C_1\) which is faster than \(C_0\) (since nested within the scope of \(C_0\)). Note that variables that refer to higher clock levels can be read or written within the scope of a faster clock: For example, the local variable \(a\) first gets the value of \(z\) which is written back in another substep to the output \(y\). However, the example also illustrates data dependencies in the following way: both threads are coupled by \(z\) which is written in clock domain \(C_2\) and which is read in clock domain \(C_1\). Since \(z\) is declared at a higher level, its value remains constant for several substeps, therefore the value read by thread of \(C_1\) must be the same written by the thread of \(C_2\). Thus, the first two substeps of \(C_2\) must be executed before the first step of \(C_1\) can be executed.

Hence, subclocks can be used to introduce a limited amount of (internal) asynchrony which can be used for a more efficient execution of the programs and to avoid the effect of over-synchronization. In addition, more flexibility is provided to the developer, who is no longer limited by the single clock that only allows the abstraction of micro steps to macro steps. Instead, programmers can now make use of a clock hierarchy that allows to refine the basic clocks into a hierarchy of subclocks.

3. Intermediate Format

Our Averest system\(^2\) contains a compiler for the synchronous language Quartz. To this end, we make use of the Averest Intermediate Format (AIF) which serves as a bridge between our target-independent compiler, and various analysis and synthesis tools customized for different architectural classes.

This intermediate format is based on synchronous guarded actions [8] to represent the behavior of a system. Guarded actions turned out to be well-suited to eliminate the complex interaction of statements of the source language on the one hand while preserving the synchronous semantics and allowing efficient analysis and generation of hardware and software code on the other hand. A guarded action \((\gamma \Rightarrow A)\) consists of the guard \(\gamma\) which is a boolean expression that

\(^1\) This example also illustrates that this kind of up-sampling is different to the one used in Signal. There, it is not possible to refine an instantaneous computation (e.g., a function node) by up-sampling a sequential one.

\(^2\) http://www.averest.org

\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
module \textit{GCD} (nat \texttt{?a}, \texttt{?b}, \texttt{!gcd}) \{
\begin{minipage}{0.9\textwidth}
\begin{verbatim}
clock(C_1) \{  
  int x = a, y = b;  
  while(x > 0) \{
    if(x \geq y)
      next(x) = x - y;
    else
      next(y) = y - x;
    \ell_1 : pause(C_1);  
  \}
  \}
\end{verbatim}
\end{minipage}
\(gcd = y;\)
\}
\textbf{(a) Greatest Common Divisor}
\end{minipage}
\begin{minipage}{0.45\textwidth}
module \textit{parallel} (nat \texttt{?x}, \texttt{!y}) \{
\begin{minipage}{0.9\textwidth}
\begin{verbatim}
clock(C_1) \{  
  \texttt{nat} z;  
  \ell_1 : pause(C_1);  
  \texttt{a} = z; 
  y = a; 
  \ell_1 : pause;  
}\end{verbatim}
\end{minipage}
\begin{minipage}{0.9\textwidth}
\begin{verbatim}
clock(C_2) \{  
  \texttt{nat} \texttt{a,b};  
  \ell_{21} : pause(C_2);  
  z = x;  
  \ell_{22} : pause(C_2);  
  \ell_2 : pause;  
}\end{verbatim}
\end{minipage}
\textbf{(b) Independent Clocks with Data Dependency}
\end{minipage}
\caption{Examples: Greatest Common Divisor and Parallel Threads}
\end{figure}
leads to the execution of the action \( A \). Guarded actions are computed for the data flow and the control flow of a given program. For the data flow, each assignment that occurs in the original program leads to one or more guarded actions. Hence, they are encoded by immediate actions of the form \( \langle \gamma \Rightarrow x = \tau \rangle \) or delayed actions of the form \( \langle \gamma \Rightarrow \text{next}(x) = \tau \rangle \). To encode the control flow, every control flow location in the source code (i.e. where a macro step can end) is endowed with a Boolean variable that is called the label of that control flow location. If the control flow moves to such a label \( \ell \), a delayed action sets \( \ell \) to true for the next step.

The semantics of the intermediate format is as follows. In contrast to traditional guarded commands [10, 11, 18], our guarded actions follow the synchronous model of computation. In each macro step, all activities refer to the same point of time, i.e. the evaluation of all expressions contained in the guarded actions refer to the same variable environment which may be partially defined, and which may be extended when more values become known due to the execution of guarded actions in the causal order following the data dependencies. If the guard of an immediate action is true, the right-hand side \( \tau \) is evaluated to determine the value of variable \( x \) in the current macro step, while a delayed action defers the update to the following step.

Similar to Quartz programs, the AIF description adds an implicit reaction to absence. If no action has determined the value of the variable in the current macro step (obviously, this is the case iff the guards of all immediate assignments in the current step and the guards of all delayed assignments in the preceding step of a variable are false), then a variable either gets a default value or stores its previous value, depending on the declaration of the variable: event variables are reset to a default value while memorized variables store their value of the previous step.

In addition to the description of the behavior by guarded actions and absence reactions, AIF contains more information like the declaration of variables and the input/output interface of the described synchronous system. Additionally, information to support modularity, and some other aspects such as specifications are covered by the intermediate format which is however not needed in this paper.

Apparently, AIF uses the same model of computation as pure Quartz, i.e. both languages divide the execution of the program into macro and micro steps. In particular, both languages do not support other clocks than the basic clock. After introducing clock refinement in the source language as described in Section 2, we also decided to reflect this extension in the intermediate format by having variables and synchronous guarded actions based on several clocks, for several reasons: In general, it is not always possible to compile Quartz programs with subclocks to the intermediate format without subclocks without changing the externally visible behavior. An example program which cannot be translated into a single-clock version with the same behavior is given in Figure 3 (a).

Moreover, even in the case where a translation is possible, the compilation would be forced to determine a particular alignment of subblocks which would eliminate the freedom we wanted to introduce so that the code generator can make use of it. Analysis tools and code generators that are later on employed would then consider a derived single-clocked synchronous system that would again be over-synchronized.

Hence, we extend the structure of the intermediate format as follows: for each variable, we additionally store its clock. In particular, the Boolean control flow labels are also bound to a clock. Additionally, the intermediate format stores the dependencies between clocks, i.e. its stores the clock tree of the system.

The semantics of the intermediate format has to be adapted accordingly. Due to the synchronous model of computation, all actions in every step are evaluated at once, and thereafter a transition to the following step takes place. In the context of several clocks, different kinds of steps must be distinguished. In the source code, we make this distinction by considering the least clock of the pause statements that are reached. In the intermediate format, this is similarly accomplished by choosing the least clock of the labels that are set by delayed actions. Note that there might be a choice, since several unrelated clocks can be enabled at the same time (see Figure 3 (b)). As for the source-code level which has been discussed in the previous section, this choice is again only constrained by potential data dependencies. Thus, transitions must not be enabled if their execution involves still unknown variables. For more details, please refer to the formal semantics given in [15].

As it becomes obvious, the extended intermediate format now follows the same model of computation of the extended source language so that a translation becomes possible which eliminates all complex statements of the source language, while preserving all semantic characteristics. The following section will now show how the compilation works in detail.

4. Compilation

4.1. Surface and Depth

The previous section presented the extended intermediate format which will be used in the following as the target of our compilation. As the correct compilation of synchronous programs is known to be a challenging problem (it has to face problems such as schizophrenia problems and the difficult interaction of nested and concurrent preemption statements), we aim at reusing our existing compilation algorithm [8, 24] as far as possible.

In particular, we want to keep the overall compilation scheme which is based on the separation of the surface and the depth of Quartz statements. Thereby, the surface of a
statement or program is the part that is executed in the first macro step of the execution, while its depth is the part that is executed in any later step. Since pause statements may be contained in conditional statements (see $\ell_1$ in Figure 4), the surface and the depth of a statement may overlap depending on data-dependent conditions.

This distinction is the key to compile complex statements of imperative synchronous languages such as local variables, abortion and suspension statements. All of them have in common that the behavior of the surface and depth must be treated separately. For instance, a schizophrenic local variable requires the replication of the surface of its scope [8, 24, 25], and several different actions may be generated for the same program statements, depending on the location from which it has been reached.

It is reasonable to use the existing compile algorithm as a starting point for the compilation of the Quartz programs with subclocks. Since there are now several clocks, surface and depth parts are now defined relative to these clocks. Therefore, we make the following definition which is compatible with the previous single-clock variant: The surface with respect to a clock $C$ is the part of the program which is executed before the next pause statement at level $C$ (or at a higher one) occurs. The depth with respect to a clock $C$ is the part of the program which is executed after the next pause statement at level $C$.

Figure 4 illustrates this definition: for the basic clock $C_0$, we have an overlap of surface and depth due to the if-statement. The depth according to $C_0$ contains every statement potentially executed after the first step wrt. the clock $C_0$. Inside the block of clock $C_1$, the surface according to $C_1$ is of interest. Note that it does not make sense to compute the surface according to $C_1$ for the whole code fragment, because the clock $C_1$ is just visible inside the block. The surface of $C_1$ stops at the first reached pause of $C_1$. The depth of $C_1$ starts after it.

If we have another look at the surface of $C_0$, we see that a step of $C_0$ will not end at label $\ell_2$ and will rather pass it, since this is a pause referring to a lower clock. Therefore, the control flow of the lower clock $C_1$ is contained in the surface of the higher one. This situation is different to single-clocked systems, where the surface is purely combinational and does not contain pause statements. This has some interesting effects on preemptions and schizophrenia, as will be discussed below.

In the following subsections, we will explain our preliminary compilation algorithm. Although control flow and data flow are computed by a single recursive traversal over the program structure in our implementation, we will present both parts separately for the sake of a clearer presentation. Thereby, we will first focus on the control flow which is the more interesting part with respect to our language extension. Then, we show how the data flow is integrated in the compilation scheme.

4.2. Control Flow

As already pointed out in Section 3, every position in the Quartz program where the control flow can rest is endowed with a Boolean location variable that is called the label of that position. These labels are set when the execution reaches the corresponding control flow location. Similar to local variables, also labels refer to clocks, depending on the clock level which ends at the label. This information can be directly derived from the source code, since it just corresponds to the clock parameter $C$ of the pause($C$) statement.

The task of the compilation of the control flow is the extraction of guarded actions that represent the movement between the labels, i.e. guarded actions that determine the labels that are active in the following step depending on the currently active labels. Hence, we want to create a set of guarded actions of the form $(\gamma \Rightarrow \text{next}(\ell) = \text{true})$. Thereby, the action expresses that $\ell$ will be reached in the following step under current condition $\gamma$. Note again that labels are automatically reset to false when no action sets them in the previous step. This is accomplished by declaring them as ‘event’ variables.

Our compilation procedure implements a traversal over the program structure to compile the program step by step. Thereby, it considers the surface and depth of a statement separately by two functions CompileSurface and CompileDepth. These functions expect the following arguments:

- clock describes the clock the current surface/depth is compiled for. This is normally the lowest clock whose scope contains the current statement. However, the compilation of the surface of a statement clock($C$) S
requires to compile the surface of \( S \) wrt. to the current clock and also wrt. to the new clock \( C \).

- \( \text{strt}_S \) is the current activation condition which must hold to reach the current position in the source code. It is used to build the activation conditions for the actions.
- \( \text{abrt}_S \) are the current abortion conditions which are derived from the surrounding abort blocks. Since we need to track them for each clock level separately, we use a map that assigns each clock level an abortion condition (see also next section). In the following, we use the notation \( \text{abrt}_S[C] \) to access an expression stored by map \( \text{abrt}_S \) for the clock \( C \).
- Similarly, \( \text{susp}_S \) describes the current suspension conditions of the surrounding suspend blocks. Also a map is used to assign a suspension condition for every available clock level. Thereby, priorities of the suspension and abortion conditions have to be respected, where precondition exemptions of higher clocks have a higher priority.
- \( \text{strg}_S \) describes whether a possible abortion or suspension is strong or weak, i.e. whether the current step is preempted at its beginning or at its end. It is needed for the distinction of strong and weak abortion. In contrast to \( \text{abrt}_S \), the different clock levels are not distinguished here, because strong abortion takes place every time this condition holds. There is no need to relate \( \text{strg}_S \) to different clocks.

Additionally, the compilation procedures return the following Boolean conditions which are used by the calls of the compilation procedure in surrounding statements:

- \( \text{inst}_S \) holds if the execution of \( S \) is instantaneous, i.e. if only micro steps are executed in \( S \). There is no distinction between instantaneity at different clock levels. Note that \( \text{inst}_S \) does not depend on labels in \( S \), because it is assumed that the control flow is still outside \( S \).
- \( \text{insd}_S \) holds iff the control flow currently rests in a statement or not. It is the disjunction of all labels of the considered clock level of statement \( S \). Thus, if there is one of its labels set, the control flow rests inside.
- \( \text{term}_S \) holds iff the control flow is currently in statement \( S \) and it terminates. The termination of a statement can be influenced by lower clock levels. Therefore, it cannot be determined for each clock level in a straightforward way.

Figure 5 (a) shows a code fragment consisting of a sequence of \( \text{pause} \) statements at different clock levels, where one of them is contained in an \( \text{if} \)-statement. Delayed guarded actions are created for each \( \text{pause} \) statement with guards reflecting the condition when it can be reached in a step.

Thus, the guard of such a guarded action is generally \( \text{strt}_S \wedge \neg(\text{abrt}_S[C] \lor \text{susp}_S[C]) \). Since the example does not contain any abort or suspend statement, we can ignore these conditions for this example.

The guarded actions created for the example are shown in Figure 5 (b). One would expect that the start condition for the first guarded action is just \( \ell_1 \), but this would lead to a wrong behavior: Assume that the guarded action \( (\ell_3 \Rightarrow \text{next}(\ell_2) = \text{true}) \) sets the label \( \ell_2 \) to true for the following step. After a tick of clock \( C_1 \), the value of \( \ell_1 \) is still true, because it is declared at a higher level. It would enable \( \ell_2 \) again for the next substep. This issue arises when a label of a higher clock starts a sequence of substeps, then the higher label must trigger the first step once. Therefore, the \( \text{strt}_S \) condition is built from the higher label and the condition that the control flow is still not at the lower level, as the expression in the example shows. Transitions from one label to another of the same clock level work as expected. Similarly, a transition to a higher level is straightforward.

The second example given in Figure 6 (a) addresses the compilation of nested abort statements of different clocks. Weak abortion takes place at the end of a step, i.e. the data flow and the lower steps are executed but not the control flow of the level of which the abortion takes places. The guarded actions which are created for the control flow for this code fragment are shown in Figure 6 (b). Note that a control flow action is created with the guard \( \text{strt}_S \wedge \neg(\text{abrt}_S[C] \lor \text{susp}_S[C]) \). The transition from \( \ell_1 \) to \( \ell_2 \) is never aborted since the control flow must be already inside the inner abort statement. The transition from \( \ell_2 \) to \( \ell_3 \) can only be aborted from the inner abort statement because this is a control flow that refers to the lower level. Finally, the transition from \( \ell_3 \) to \( \ell_4 \) can be aborted by both abort conditions since the step at the higher level also ends at this position. This examples

---

3 In general, a clock can be refined by more than one clock due to nested statements \( \text{clock}(C) \). However, every statement of a program is contained in only one branch of the clock tree.
Weird abort {  
  ℓ₁ : pause;
  clock(C₁) {  
    weak abort {  
      ℓ₂ : pause(C₁);  
      ℓ₃ : pause(C₁);  
      ℓ₄ : pause;
    } when(β);
  }
} when(α);

(a) Source Code

ℓ₁ ∧ ¬ℓ₂ ∧ ¬ℓ₃ ⇒ next(ℓ₂) = true
ℓ₂ ∧ ¬β ⇒ next(ℓ₁) = true
ℓ₃ ∧ ¬β ∧ ¬α ⇒ next(ℓ₄) = true

(a) Control Flow

Figure 6. Abort Conditions of Different Clocks

thereby illustrates why it is necessary to have abortion and suspension conditions at different levels.

4.3. Data Flow

With the compilation of the control flow, we obtain guarded actions that represent the program structure. This skeleton is complemented by the data flow which consists of the assignments in the source program. They are directly translated to immediate and delayed actions in the intermediate code. Thereby, the compilation algorithm for pure Quartz only has to extract their guards which corresponds to the activation condition \(σ\) when reaching the assignment by a recursive call of the compilation. The structure of the compilation algorithm which compiles the overlapping surface and depth separately, ensures that all situations which lead to the execution of an assignment will be found and added as separate guarded actions to the intermediate code.

Fortunately, the extension to several clock layers does not require noteworthy changes for the compilation of the data flow. It is still the case that the assignments in the program are translated to guarded actions whose guards are the activation condition of the recursive call, i.e. it depends on the control flow label from where the action can be activated. Clock levels do not have any influence on these labels, since the action should be executed in the (sub)step when the control flow rests at this position. Hence, there are no changes in the existing compilation algorithm for our extension as long as we follow the general compilation scheme presented in the previous section.

In order to understand that the existing calling scheme is sufficient for a correct extraction of the data flow actions, consider preemptions which are commonly seen as the most critical statements for any synchronous compilation algorithm.

First assume that we compile a strong preemption such as \(abort S \ when(σ)\). The actions in the depth of \(S\) must be guarded by the abortion condition \(σ\), while actions in the surface must not be guarded. A closer look at the compile scheme reveals that it respects this condition. When entering an abortion statement, the abortion conditions carried by the parameter \(abrt_S\) remain unchanged, while \(CompileDepth\) adds \(σ\) to \(abrt_S\) and \(strg_S\). The latter one is then used for the creation of the activation condition \(strg_S\) in the compilation of the sequence which is finally used as the guard for all actions.

Now assume that we compile a weak preemption such as \(weak \ abort S \ when(σ)\). In that case, \(σ\) is not added to \(strg_S\) but only to \(abrt_S\). By the sequence rule, this information is propagated to the compilation of the actions. If we now consider abortions at several clock layers, we see again that we need to store the condition for each layer separately. Weak abortion always takes place at the end of a ‘step’ which is relative to the clock, i.e., if the abortion condition \(σ\) holds, we must exit \(S\) at the next \(pause\) of the level on which the abort starts, and all actions fire until we reach that point. Due to this reason, the abortion conditions are stored separately, and the compilation of the \(pause(C)\) always considers the conditions at level \(C\) and not on the current position.

Another problem is the translation of schizophrenic local variables. If they occur in a loop, they generally require that the surface of the loop must be replicated since old and new scopes may contain different values (which cannot be stored in a single variable) and may overlap in a (macro) step. The structure of the compilation algorithm also ensures for this statement that exactly the right part of the scope is replicated, i.e. the surface with respect to the current clock at the given position. Again, lower clock levels and their control flow are also duplicated which automatically results to schizophrenic labels which are not known from pure Quartz.

However, our experience has shown that schizophrenic local variables cause particularly subtle problems. Therefore, we do not claim that the presented solution is really sufficient for their correct translation. This statement should be only made after an elaborate formal verification similar to the one for the single-clock case given in [25].

5. Example

In this section, we consider a more complicated example which has been translated by our preliminary compiler for the subclock extension of Quartz. The example is shown in Figure 7 (a). The module consists of two threads running in parallel. To increase the readability, the threads are drawn besides each other (instead of below each other as in the real source code). The two unrelated clocks \(C₁\) and \(C₂\) are
module Sample (bool ?x, ?y, a?, nat !o)
{
    bool z;
    weak abort {
        \ell_1 : pause;
    }
    clock(C_2) {
        nat d;
        d = 0;
        abort {
            next(d) = d + 1;
            \ell_2 : pause(C_2);
            next(d) = d + 1;
            \ell_3 : pause;
        }
    }
    clock(C_1) {
        bool c;
        c = x;
        \ell_11 : pause(C_1);
        if(y)
            \ell_2 : pause;
        z = c;
        \ell_4 : pause;
    }
    } when(a);
    next(z) = x \land y;
}

(a) Source Code

\ell_1 \Rightarrow next(\ell_1) = true
\ell_1 \land \neg \ell_11 \Rightarrow next(\ell_11) = true
\ell_11 \land y \land \neg a \Rightarrow next(\ell_3) = true
\ell_11 \land y \land a \Rightarrow next(\ell_4) = true
\ell_3 \land \neg a \Rightarrow next(\ell_4) = true
\ell_1 \land (\ell_21 \lor \ell_22 \lor \ell_23) \Rightarrow next(\ell_21) = true
\ell_21 \land \neg z \Rightarrow next(\ell_22) = true
\ell_22 \lor (\ell_21 \lor \ell_22 \lor \ell_23) \Rightarrow next(\ell_23) = true
\ell_23 \land \neg a \Rightarrow next(\ell_2) = true

(b) Control Flow

\ell_3 \land \neg \ell_11 \Rightarrow c = x
\ell_11 \land \neg y \Rightarrow z = c
\ell_11 \land y \Rightarrow z = c
\ell_1 \land (\ell_21 \lor \ell_22 \lor \ell_23) \Rightarrow d = 0
\ell_1 \land (\ell_21 \lor \ell_22 \lor \ell_23) \Rightarrow next(d) = d + 1
\ell_21 \land \neg z \Rightarrow next(d) = d + 1
\ell_22 \land \neg z \Rightarrow next(d) = d + 1
\ell_23 \land \neg a \Rightarrow o = d
\ell_4 \land \ell_2 \lor \ell_4 \land \neg \ell_2 \lor \ell_2 \land \neg (\ell_3 \lor \ell_4) \lor (\ell_1 \lor \ell_2 \lor \ell_3 \lor \ell_4) \land a \Rightarrow next(z) = x \land y

(c) Data Flow

\ell_0
\ell_1
\ell_11
\ell_3
\ell_4
\ell_21
\ell_22
\ell_23
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4
\ell_11
\ell_3
\ell_4

Figure 7. Compilation Example

declared in the threads and refine the module clock C_0. The threads are surrounded by a weak abort statement which additionally contains the \ell_1 : pause at the beginning of the block. Therefore, after the first step, the abortion can take place, when the condition a which is an input holds. The left thread performs an assignment to the local variable c and finally sets the variable z. The second thread declares a local variable d which is increased for some substeps. However, the incrementation is aborted by the condition z which is set by the first thread. Since, the abortion is defined at the level of C_2, it cannot take place in the first C_2 step, but in the second one. Therefore, the first incrementation is definitely executed. At the end, the output o is assigned by the value of d. At the end of the whole module which is reached if both threads terminate or if they are aborted by the surrounding abortion statement, the assignment next(z) = x \land y is executed.

Our compiler generates the actions for the control flow and data flow shown in Figure 7 (b) and (c), respectively. Additionally, it extracts the declarations of the variables and the clock tree: variable c is declared on clock C_1, variable d on C_2, and the unrelated clocks C_1, C_2 are both subclocks of the module clock C_0. The label \ell_0 is created automatically as start condition for the whole module. It must be set to trigger a start of the module.

The action \langle \ell_1 \land \neg \ell_11 \Rightarrow next(\ell_11) = true \rangle illustrates the start condition for the first substep. If we assume \ell_1 as guard of this action, it would be executed in every substep, because \ell_1 is defined at a higher level. However, the action should only be executed for the first substep, therefore the guard takes care that the control flow is not still in a following substep.

There are two actions generated for the assignment z = c, but there is just one assignment in the source code. Some points in the source code can be reached by surface and depth, and therefore different actions may be created. As already explained, the distinction of clocks, makes some positions reachable from different positions by different clock levels. In the example, the assignment z = c can be executed when starting from \ell_3 or \ell_11. The actions may be combined by some additional analysis, but the structure of the compile procedure treats them separately.

The effect of the surrounding weak abort statement is also interesting. The condition a is just added to control flow actions which activate a label at the level of C_0. In the single-clocked case, weak abortion executes the data flow, but aborts the control flow. For refined clocks, this means that the substeps are executed, but the whole step is aborted. Therefore, lower steps are not aborted until the level of C_0 is reached again.

6. Conclusions

After introducing the concept of clock refinement to imperative synchronous languages, we made the next step towards compilation of the proposed extensions. This paper presented an extended intermediate format and a preliminary compilation algorithm which translates a program of the
extended source language to that new representation. The complex interaction of statements is removed after the translation to the intermediate language, while the semantics of the synchronous language with refined clocks is kept so that further analysis and synthesis procedures can take full advantage of the intermediate representation.

For trustworthy use of the new compile algorithm, further work is still needed. After an elaborate verification of the presented compilation algorithm with respect to schizophrenic local variables and its extension to modular compilation, several related issues must be tackled: The causality and reactivity of the system must be checked before synthesis, as it is the case for any synchronous system. Therefore, existing techniques must also be extended to handle refined clocks. Moreover, new synthesis procedures must be developed to create efficient code that exploits the additional freedom introduced by refined clocks.

References


