EURASIP Journal on Embedded Systems
a SpringerOpen Journal

REVIEW Open Access

Clock refinement in imperative synchronous languages

Mike Gemünde*, Jens Brandt and Klaus Schneider

Abstract

The synchronous model of computation divides the program execution into a sequence of logical steps. On the one hand, this view simplifies many analyses and synthesis procedures, but on the other hand, it imposes restrictions on the modeling and optimization of systems. In this article, we introduce refined clocks in imperative synchronous languages to overcome these restrictions while still preserving important properties of the basic model. We first present the idea in detail and motivate various design decisions with respect to the language extension. Then, we sketch all the adaptations needed in the design flow to support refined clocks.

1 Review

Synchronous languages [1] such as Esterel [2], Lustre [3], or Quartz [4] have been proposed for the development of safety-critical embedded systems. They are based on a convenient programming model, which allows one to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can directly be executed on simple micro-controllers without having the need to use complex operating systems. In addition, synchronous programs can straightforwardly be translated to hardware circuits [4-6], which makes synchronous languages attractive for the use in hardware–software co-design. Furthermore, the concise formal semantics of synchronous languages is the basis for formal verification of the correctness of the programs as well as of the used compilers [7-10]. Finally, since macro steps consist of only finitely many micro steps whose number is known at compile-time, one can determine tight bounds on the reaction time by a simplified worst-case execution time analysis [11-14].

All these advantages are due to the underlying *synchronous model of computation* [1], which divides the execution of programs into *micro and macro steps*, where variables change synchronously only between macro steps and remain constant during micro steps. The partitioning into micro and macro steps is explicitly given by the programmer, and the micro steps are executed in a causal ordering so that there are no read-after-write conflicts [7,15]. As a consequence, all threads of a program run in lockstep: they execute the micro steps of their current macro steps in the common global variable environment, and therefore automatically synchronize at the end of the macro step.

Obviously, the synchronous model of computation enforces deterministic concurrency, which has many advantages in system design, e.g., to avoid Heisenbugs [16] and to allow compile-time analyses, e.g., on WCET. At the same time, however, it imposes tight restrictions on modeling possibilities, since there is no means to express the independence of threads in certain program locations. This phenomenon—where synchronous lockstep execution of threads is enforced even though it is not necessary—is often referred to as *over-synchronization*. Over-synchronization occurs quite frequently, since the input signals of a system usually have different rates, and even signals of the same rate do not necessarily need to be synchronized if there are no data dependencies among them. While a static clock and data-flow analysis may be able to detect the dependencies to desynchronize such programs [17], adding an explicit notion of independence makes it possible for compilers to create desynchronized code without sophisticated and expensive analyses.

Another deficiency of the synchronous model is its inflexibility with respect to temporal changes. Modifications of the temporal behavior of a component may be problematic since they can endanger the global

*Correspondence: gemuende@cs.uni-kl.de
Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany

behavior of the entire system. For this reason, design methods such as latency-insensitive [18] design or synchronous elastic systems [19,20] have been developed to maintain the synchronous computation between modules in case the timing of one of the modules is changed.

Another desirable feature for imperative synchronous languages, which requires further temporal abstraction layers, is *function calls*, which must be executed within a micro step: For example, assume that the *greatest common divisor (GCD)* of two integers is required in a program expression. As such non-primitive recursive functions require data-dependent loops, it is not possible to implement them as micro steps of a macro step since the number of micro steps depends on values. Executing parallel function calls imposes a lot of problems since lazy evaluation or other kinds of code optimization destroy the temporal behavior. Wrapping functions into module calls causes even more problems, since the function parameters should be constant during function evaluation, which must explicitly be enforced by the caller. A true function interface would guarantee this by definition.

All these problems can be solved by providing a hierarchy of clocks in the synchronous system that does not only allow to combine macro steps to a larger step of a slower clock, but that also allows one to refine the base clock into different faster clocks. Thereby, it is possible to explicitly *describe* the point of time, when synchronization should happen, independent of the number of steps that have been passed or that are needed for a calculation. The refinement of the base clock in a module is particularly attractive since it retains the external input/output behavior. Thereby, it is possible to replace a code segment by another one having a different temporal behavior. For instance, it becomes possible to exchange components with functionally equivalent ones running at higher clock speeds. Obviously, refinements make component-based design much more flexible.

The rest of this article is structured as follows. Section 2 briefly introduces the imperative synchronous language Quartz, which serves as the starting point for our extension which is presented in Section 3. Section 4 sketches a formal semantics for the extension. Section 5 gives details about the compilation of our extended Quartz to a new intermediate format, and from there finally to hardware and software. Section 6 finally discusses related work before we draw some conclusions in Section 7.

2 The synchronous language Quartz

This section introduces the synchronous model of computation with the example of the imperative synchronous language Quartz. The synchronous model of computation [1,21] divides the execution of a program into a sequence of macro steps [22]. In each macro step, the system reads the inputs, performs some computation and finally produces the outputs. In theory, the semantics assumes that the outputs are computed *in zero-time*. In practice, the execution implicitly follows the data dependencies between the micro steps, and outputs have to be computed *in bounded time* for the given application. Thus, the synchronous model of computation abstracts from communication and computation delays and considers only the dependencies of the data. A consequence of this abstraction is that each variable has a designated value in each macro step.

2.1 Statements

The imperative synchronous language Quartz implements the synchronous model of computation by means of the **pause** statement. While all other primitive statements do not take time (in terms of macro steps), a **pause** marks the end of a macro step and consumes one logical unit of time. Thus, the behavior of a whole macro step is defined by all actions between two consecutive **pause** statements. Parallel threads run in lock-step: their macro steps are executed synchronously, and the statement in both are scheduled according to the data dependencies so that all variables have a unique well-defined value in the macro step.

We illustrate the synchronous model of computation by a simple example shown in Figure 1a. It takes two inputs `i1`, `i2`, produces two outputs `o1`, `o2`, and has one local variable `x`. Every **pause** statement is annotated with a label for better identification. Figure 1b shows an execution of the program based on some sample input values. For space reasons, the values true and false are written as T and F in the figure. In the first macro step, the program is started (st is true) all actions before the first **pause** statement are executed. In the example, these are the assignments to `o1` and `x` which assigns the values 3 and 1 based on the given input values. There is no assignment to `o2` which therefore gets its default value 0. In the second macro step, the execution resumes from the **pause** statement with label `l1`. The label is set to true, and all other labels are false for this step. In this second step, the variables `o1`, `o2` and `x` are assigned. Since each variable has a unique value for the entire step, the value that is assigned to `o2` is used to determine the value for `o1`. Thus, the assignment to `o2` must be executed *before* the assignment to `o1`. The resulting values are shown in the table. The next macro step starts from the **pause** statement with label `l2`. Due to the **if** statement, the assignment to `o1` is not executed. Since `o1` is not set by an assignment, it stores its value from the last step.

```
module P1
  (nat ?i1,?i2,o1,o2)
{
  nat x;
  loop {
    o1 = i1 + i2;
    x  = i1;
    l1: pause;
    o1 = o2 + i1 + x;
    o2 = i2;
    x  = 2;
    l2: pause;
    if(i1 > 4)
      o1 = i1;
    o2 = i1 + o1;
    l3: pause;
  }
}
```

(a) P1 Source Code

Steps		1	2	3	4	5
Inputs	i1	1	2	3	4	5
	i2	2	4	6	8	0
Labels	st	T	F	F	F	F
	l1	F	T	F	F	T
	l2	F	F	T	F	F
	l3	F	F	F	T	F
Locals & Outputs	x	1	2	2	4	2
	o1	3	8	8	12	7
	o2	0	4	11	11	0

(b) Sample Execution Trace

Figure 1 `P1` example.

Basically, all Quartz programs can be reduced to the following set of *basic statements*, which can be used to define further macro statements as syntactic sugar:

- **`nothing`**
  This statement has no effect. It only exists for technical reasons of defining source code transformations.
- `l: pause`
  The **`pause`** marks the end of a macro step and thus also the begin of the following step.
- `x =` τ
  This form of a variable assignment is called an *immediate* assignment. It sets the value of the variable `x` for the current step to the value given by the evaluation of the expression τ.
- **`next`**`(x) =` τ
  This form of variable assignment is called an *delayed* assignment. Like the immediate assignment, it evaluates the expression τ with the values of the current step, but this value is assigned to the variable **`x`** only in the following step.
- **`do`** `...` **`while`**(σ)
  This loop statement first executes its body statement. If the body statement terminates, the condition σ is evaluated and if σ holds, the body statement is restarted in the same step. Otherwise, the loop terminates. All other loop versions can be reduced to this basic loop.
- `{ ... } || { ... }`
  The parallel statement executes both code blocks in parallel where in each step, one macro step of each block is executed. We call the two code sub-statements of the parallel statement *threads*. One can therefore also say that the threads synchronize on each **`pause`** statement that is reached. The parallel statement terminates if its last thread terminates.
- **`abort`** `...` **`when`**(σ)
  With the (*strong*) abortion statement, the execution of a code block can be aborted when the given condition holds. The abortion takes place at the *beginning* of a macro step: if the condition holds in a step, no action inside of the code block is executed.
- **`suspend`** `...` **`when`**(σ)
  With the (*strong*) suspension statement, the execution of a code block can be stopped when the given condition holds. In this case, the execution is stopped for the whole macro step and no action inside the block is executed. The execution resumes at the next macro step where the condition does not hold.

There are also other statements which are not considered here, because we do not define the extension with refined clocks for them.

2.2 Logical correctness and causality

In the synchronous model of computation, all micro steps in a macro step are executed synchronously. In theory, every variable assignment that complies to the execution can be considered as a consistent one. A program that has for each input assignment exactly one consistent assignment of all variables is called *logically correct*. We illustrate this concept with the help of the following Quartz program.

```
l1: pause;
if (x | y) {
  x = true;
} else {
  x = true;
  l2: pause;
}
y = true;
l3: pause;
```

Consider the step that starts from label l1. Assume that the variables x and y had the value false in the previous step, i. e. if no assignment sets them in the considered step they will keep their values of the previous step. In order to be logically correct, a unique variable assignment has to be found which leads to a valid execution of the program. In principle, we can check all possible variable assignments.

It is easily seen that only the assignment (x = true, y = true) is consistent. Thus, this program (or at least the considered step) is logically correct. However, for a real execution of such a program, considering all possibilities is too inefficient. Therefore, the semantics of any synchronous language, and the one of Quartz in particular, also requires a constructive execution of programs (so that the above example is not a constructive Quartz program). In Quartz, this means that actions can be only executed if all control-flow conditions contributing to their trigger can be evaluated before that action. The control-flow conditions to execute the assignments to x and y depend on the values of x and y which is not allowed. Instead it should be possible to evaluate those control-flow conditions from already known values. Checking this property statically is known as *causality analysis* [7,15,23-28] in the context of synchronous programs.

2.3 Compilation and intermediate representation

Quartz is currently the language of the Averest framework (http://www.averest.org), which contains tools for simulation, compilation, verification, and synthesis for Quartz [4]. Thereby, its compiler translates the source files to the *Averest Intermediate Format (AIF)* [29]. AIF abstracts from the complexity of the source language: difficult interactions of preemption statements or reincarnations of local variables [7,30,31] are no longer an issue. Nevertheless, AIF files contain the entire behavior of the given synchronous program, and they are therefore the central part of the target-independent analyses of various back end tools.

The intermediate format describes the behavior with the help of synchronous guarded actions [31], which 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 is of the form:

$$\gamma \Rightarrow A \tag{1}$$

where γ is called the *guard* and A is an *action*, i.e., either an immediate or a delayed assignment. The intention is that the action is executed in an instant whenever its guard holds. Thus, the data-flow actions can be collected from the source code, and the compiler determines their corresponding guards.

Guarded actions do not only represent the data-flow, i.e., assignments occurring in a program, but they are also used for the control-flow. To this end, all program labels are encoded as Boolean events (and additionally adding an implicit start label st). The control flow can then be described by actions of the form $\langle\gamma \Rightarrow \texttt{next}(\ell) = \mathsf{true}\rangle$, where γ is a condition that is responsible for moving the control flow at the next point of time to location ℓ. For instance, the guarded actions for program **P1** (see Figure 1a) are given in Figure 2.

The semantics of the intermediate format is as follows. In contrast to traditional *guarded commands* [32-34], guarded actions follow the synchronous model of computation. In each macro step, all actions refer to the same point of time, i.e., the evaluation of all expressions contained in the guarded actions refers to the same variable environment. If the guard τ of an immediate assignment $\gamma \Rightarrow x = \tau$ is true, the right-hand side τ 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 *default reaction*: if no action has determined the value of the variable in the current macro step, then a variable either gets a default value or stores its previous value, depending on the declaration of the variable (obviously, this is the case if 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 evaluated to false). Thereby, *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 default reactions, AIF contains more information such as the declaration of variables

$$
\begin{array}{rcl}
\mathtt{st} \Rightarrow \mathtt{o1} & = & \mathtt{i1} + \mathtt{i2} \\
\mathtt{st} \Rightarrow \mathtt{x} & = & \mathtt{i2} \\
\mathtt{l1} \Rightarrow \mathtt{o1} & = & \mathtt{x} + \mathtt{i1} + \mathtt{o2} \\
\mathtt{l1} \Rightarrow \mathtt{o2} & = & \mathtt{i2} \\
\mathtt{l1} \Rightarrow \mathtt{x} & = & 2 \\
\mathtt{l2} \wedge \mathtt{i1} > 4 \Rightarrow \mathtt{o1} & = & \mathtt{i1} \\
\mathtt{l2} \Rightarrow \mathtt{o2} & = & \mathtt{i1} + \mathtt{o1} \\
\mathtt{l3} \Rightarrow \mathtt{o1} & = & \mathtt{i1} + \mathtt{i2} \\
\mathtt{l3} \Rightarrow \mathtt{x} & = & \mathtt{i2} \\
\multicolumn{3}{c}{\textit{Data-Flow Actions}}
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{st} \Rightarrow \mathtt{next}\,(\mathtt{l1}) & = & \mathsf{true} \\
\mathtt{l1} \Rightarrow \mathtt{next}\,(\mathtt{l2}) & = & \mathsf{true} \\
\mathtt{l2} \Rightarrow \mathtt{next}\,(\mathtt{l3}) & = & \mathsf{true} \\
\mathtt{l3} \Rightarrow \mathtt{next}\,(\mathtt{l1}) & = & \mathsf{true} \\
\multicolumn{3}{c}{\textit{Control-Flow Actions}}
\end{array}
$$

Figure 2 Guarded actions of program `P1`.

and the input/output interface of the described synchronous system. The intermediate format contains more information (e.g., about modularity or verification), which we skip since it is not needed in this article.

The algorithm which translates a given Quartz program to guarded actions is given in [29,31], and we will only sketch its basic idea in this article. The whole procedure is split into two functions, which determine the *surface* and *depth* [7] of each statement:

- **surface**
  The surface contains the guarded actions which are executed in the macro step in which the considered statement is started.
- **depth**
  The depth contains the guarded actions which are executed in all following steps *after* the statement was started.

Thereby, the whole compilation slices the program into steps, i.e., the depth compilation makes use of the surface compilation, which traverses the abstract syntax tree (AST). The separation into surface and depth is essential for the correctness of the compilation algorithm.

3 Language extension

As already stated in Section 1, all threads of a Quartz program are based on the same timescale and therefore, they synchronize at each **`pause`** statement. If they do not communicate, then the synchronization is not necessary, but still enforced by the synchronous model of computation. This so-called *over-synchronization* is therefore an undesired side-effect of the synchronous model of computation. We present clock refinement as a solution to overcome this problem. This extension was recently proposed [35] to avoid the described effects and others. First, Section 3.1 describes the basic idea of the extension, then we illustrate the underlying time model in Section 3.2, and we finally discuss some design decisions and their consequences in Section 3.3.

3.1 Basic idea of refined clocks

The basic idea of the language extension is explained in the following with the help of two implementations of the *Euclidean Algorithm* to compute the GCD. The first variant, which is given in Figure 3a, does not use clock refinement. The module reads its two inputs *a* and *b* in the first step and assigns them to the local variables `x` and `y`. Then, the module computes iteratively the GCD of the local variables. The computation steps are separated by the **`pause`** statement with label `l1`. Each variable has a unique value in a step, and the delayed assignments set a new value to the variables for the following step. Finally, the GCD is written to the output variable **`gcd`**. Apparently, a drawback of this implementation is that the computation is spread over a number of steps. The actual number depends on the input values, and each call to this module has to take care of the consumption of time. An example execution trace for the computation of the GCD of the numbers 7 and 3 is shown in Figure 4a. The computation takes six steps and during this computation, the inputs a and b may change in principle. Thus, a calling module has to take care of the computation steps until the result is available.

The second variant, which is shown in Figure 3b, uses clock refinement. While the overall algorithm remains the same, the GCD computation is now hidden in the declaration of the local clock `C1`. The computation steps are separated by the **`pause`** statement with label `l`, which now belongs to the clock `C1`. In contrast to the first variant, the computation does not hit a **`pause`** statement of the outer clock and thus, the computation steps are not visible to the outside. As a consequence, each call to this module seems to be completed in a single step. The local variables `x` and `y` are now declared inside the local clock block and therefore, they can change their value for each

```
module GCD1
  (nat ?a, ?b, !gcd)
{

  nat x, y;
  x = a;
  y = b;
  while(x > 0) {
    if(x >= y)
      next(x) = x-y;
    else
      next(y) = y-x;
    l: pause;
  }
  gcd = y;

}
```

(a) Single Clock

```
module GCD2
  (nat ?a, ?b, !gcd)
{
  clock(C1) {
    nat x, y;
    x = a;
    y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x-y;
      else
        next(y) = y-x;
      l: pause(C1);
    }
    gcd = y;
  }
}
```

(b) Clock Refinement

Figure 3 GCD.

step of the local clock, which is crucial for the correct execution of the algorithm in this example. An example execution trace for the computation of the GCD of the numbers 7 and 3 is shown in Figure 4b. The computation for the version with a refined clock takes also six steps, but these are steps of clock C1. The computation is finished in one step of the module's clock. The variables a, b, and gcd, which are declared on the module's clock, only have one value for this base step, while the variables x and y, which are declared on clock C1, change their value for each step of clock C1. Thus, the inputs remain constant during the computation and there is only one value of the output gcd.

The trace shows even more: In the synchronous model, each variable has exactly one value for each step, and this value is valid from the beginning to the end of the step. The inputs are given from the outside and thus, they are known for the whole computation. The output gcd is computed after some substeps, but in the general view, it is valid during the whole step. An additional note should be given on the term *clock*, because it is often used for different concepts. In this case, the clock is about the description of the computation and the control-flow of the language. It is not to trigger computations from the outer environment by, e.g., a periodic signal. This distinction is considered again in Section 6.

Obviously, it is not only possible to arbitrarily nest clock declarations, but also to introduce new clocks in separate scopes. This gives rise to the clock tree of a program, which can directly be obtained from the program structure. Figure 5 gives an example: the left-hand side shows

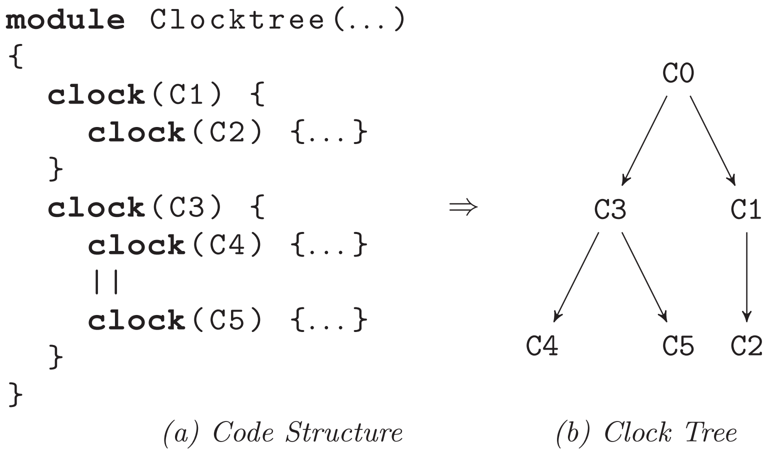
	1	2	3	4	5	6
a	7	7	7	7	7	7
b	3	3	3	3	3	3
st	T	F	F	F	F	F
l	F	T	T	T	T	T
x	7	4	1	1	1	0
y	3	3	3	2	1	1
gcd	0	0	0	0	0	1

(a) Single Clock

	1	2	3	4	5	6
a	7					
b	3					
st	T	F	F	F	F	F
l	F	T	T	T	T	T
x	7	4	1	1	1	0
y	3	3	3	2	1	1
gcd						1

(b) Clock Refinement

Figure 4 GCD traces.



```
module Clocktree(...)
{
  clock(C1) {
    clock(C2) {...}
  }
  clock(C3) {
    clock(C4) {...}
    ||
    clock(C5) {...}
  }
}
```

(a) Code Structure *(b) Clock Tree*

Figure 5 Clock tree of program.

the structure of nested clock declarations in source code, and the right-hand side shows the according clock tree, which can be derived from it.

3.2 Different views at the time model

The synchronous model abstracts time to reaction *instants*. The imperative synchronous language Quartz implements this model by *steps*, which range from a **pause** statement to another **pause** statements in the source code. Thus, in this single clock model, instants coincide with steps.

This section discusses two different interpretations of refined clocks and **pause** statements related to a particular clock. The first interpretation keeps the view that a step of the module coincides with an instant, whereas in the second interpretation new instants are introduced by the refined clocks, but these instants are not visible to the outside. We call the first one the *step view* and the second one the *instant view*. Both of them are discussed in more detail in the following. In single-clock Quartz, the following two interpretations of the **pause** statement are possible:

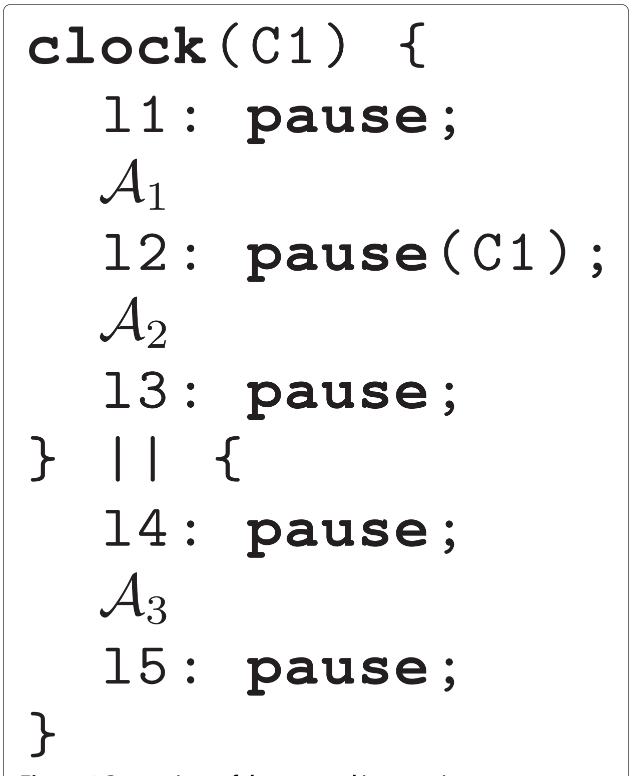
1. A step ranges from one **pause** statement to a **pause** statement and everything in between defines the behavior of the execution instant. Thus, the **pause** statement separates two steps.
2. The program execution *waits* at a **pause** statement for a clock tick. When it occurs, the program is executed until the next **pause** statement is reached and the execution stops and waits for the next clock tick to occur. It can be seen as a special kind of the **await** statement that waits for clocks.

The distinction between the above two views might appear artificial and irrelevant, so one might say that both views are the same. This is mostly true for the single clock case, where both views coincide, but when refined clocks come into play, both views become different:

1. A step of a clock ranges from one **pause** statement of this clock to another one. In between, **pause** statements of a lower clock can occur, which hierarchically divide the step into substeps.
2. The execution waits at a **pause** statement for the occurrence of the clock the **pause** belongs to. Then the execution proceeds to the next **pause** statement and waits again. This view introduces new instances to the execution, but there is no forced synchronization in each step, because different threads may wait for different clocks.

The difference of both interpretations for refined clocks is illustrated by a code example in Figure 6. Note that the refined clock `C1` is locally declared in the first thread. Assume that the control-flow is currently at labels `l1` and `l4`. Then, we compare both interpretations:

1. According to the first interpretation, everything between two **pause** statements of the same clock belongs to a step of this clock. We are interested in steps of the module's base clock `C0`, which is not explicitly declared. In the first thread, this step ends at the **pause** statement with label `l3`. In the second thread, this step ends at label `l5`. Both threads execute one step synchronously, and thus, all parts of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ referring to the module clock `C0` are executed together, regardless of the separation of $\mathcal{A}_1$ and $\mathcal{A}_2$ due to `C1`.
2. According to the second interpretation, the program waits for a clock tick of the clocks given as the argument of the **pause** statements, and the



```
clock(C1) {
   l1: pause;
   $\mathcal{A}_1$
   l2: pause(C1);
   $\mathcal{A}_2$
   l3: pause;
} || {
   l4: pause;
   $\mathcal{A}_3$
   l5: pause;
}
```

Figure 6 Comparison of the step and instant view.

execution proceeds to the next **pause** statements, waiting there for the next tick. When a tick of the module clock occurs, $\mathcal{A}_1$ and $\mathcal{A}_3$ are executed synchronously, and the labels l2 and l5 are reached, where the execution waits for the next tick, which can only be C1. The execution proceeds with $\mathcal{A}_2$, and the first threads finally reaches label **l3** while the second threads simply waits at label **l5** for the next tick of **C0**.

In the first interpretation, $\mathcal{A}_2$ and $\mathcal{A}_3$ are executed synchronously and thus, $\mathcal{A}_3$ can depend on $\mathcal{A}_2$. In the second interpretation, both blocks are explicitly ordered, and $\mathcal{A}_3$ is executed before $\mathcal{A}_2$ so that $\mathcal{A}_3$ cannot depend on $\mathcal{A}_2$. Thus, the second interpretation is a more operational style of description, whereas the first one can be seen as a more declarative way.

The intention of this extension is to provide an (operational) executable model, which lead to the conclusion that the second view is taken. This decision can be justified with constructive semantics for Quartz and Esterel: not each logically correct program is considered a *good* one for execution. Even if some actions are executed in the same instant, they also can anyway depend on each other by an causal order. The discussion in the following section will also confirm this choice.

3.3 Refined clocks in Quartz programs

This section discusses several design decisions for our extension and their effects, in particular to the dataflow of Quartz programs. As we will see, the most general variant of the extension has to deal with many problems, which makes it too inefficient for practical examples (and probably too complex for developers). The result of our discussion are several restrictions which limit the set of valid programs to a reasonable subset, where the additional complexity is manageable. This approach is similar to the constructive semantics of Quartz which does not allow all *logically correct* programs.

In the following, we use the notation $\mathcal{A}_i$ to identify some arbitrary actions (assignments) in the source code. Direct dependencies are denoted as $\mathcal{A}_1 \xrightarrow[x]{C} \mathcal{A}_2$, which means that there is an action in $\mathcal{A}_1$ that writes the variable x of clock C that is read by an action in $\mathcal{A}_2$. Thus, $\mathcal{A}_2$ cannot be executed before $\mathcal{A}_1$.

3.3.1 Backward data flow

From the semantical point of view, substeps can be seen as micro steps of the higher clock level. In principle, they are executed simultaneously in a single step based on the higher clock. However, if we take a finer grained view, we notice that the substeps are actually executed sequentially. Without any additional constraints, this has the consequence that information can flow backwards in the program across substeps since a variable on the higher level does not change throughout the whole (super) step. Consider the following fragment of code as an example:

```
clock(C1) {
  l1: pause;
  $\mathcal{A}_1$
  l2: pause(C1);
  $\mathcal{A}_2$
  l3: pause;
}
```

This code fragment basically contains one step of clock C0, which starts at l1 and ends at l3. This step is divided into two substeps of clock C1, where the first substep executes the actions $\mathcal{A}_1$ and in the second one the actions $\mathcal{A}_2$. The following cases of dependencies can occur for the above example:

- $\mathcal{A}_1 \xrightarrow[x]{C0} \mathcal{A}_2$
  The variable x of clock C0 is written by an action in $\mathcal{A}_1$ and read by an action in $\mathcal{A}_2$. Since x refers to clock C0, it has exactly one value for the whole step from l1 to l3. This dependency seems to be no problem, because both steps of clock C1 can be executed in the right order.

- $\mathcal{A}_2 \xrightarrow[\texttt{x}]{\texttt{C0}} \mathcal{A}_1$
  The variable x of clock C0 is written by an action in $\mathcal{A}_2$ and read by an action in $\mathcal{A}_1$. Since x refers to clock C0, it has exactly one value for the whole step from l1 to l3 and value of x is needed for the execution of $\mathcal{A}_1$. However, an implicit execution order of the both steps is given by their ordering in source code. Thus, the information flows backwards due to the substeps.

It seems to be possible to solve the second dependency for the second case in the above example by a simple analysis. However, the examples can be much more complex, since control-flow introduces the question whether an action is finally reached or not.

```
clock(C1) {
 l1: pause;
 $\mathcal{A}_1$
 while($\gamma$) {
   $\mathcal{A}_2$
   l2: pause(C1);
}
 $\mathcal{A}_3$
 l3: pause;
}
```

Assume that we have the dependencies $\mathcal{A}_3 \xrightarrow[\texttt{x}]{\texttt{C0}} \mathcal{A}_1, \mathcal{A}_1 \xrightarrow[\texttt{y}]{\texttt{C0}} \gamma, \mathcal{A}_2 \xrightarrow[\texttt{z}]{\texttt{C1}} \mathcal{A}_3$ of the actions for example above. The condition γ of the loop depends on a variable y of clock C0 but the computation of the variable in $\mathcal{A}_1$ depends on another variable x, which is computed at the end of the step. In addition, the loop must terminate to reach the end of the step of clock C0. The variable z, which belongs to clock C1, is changed in the loop and finally it is used to compute x. Thus, the whole loop has to be iterated to its end to check whether the loop has to be entered or not.

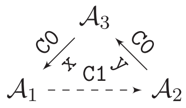
The example illustrates two points: First, introducing refined clocks without additional constraints would require an expensive (reachability) analysis, and second, it seems to be unnatural to the developer since **pause** statements of clock C1 impose a sequential order of substeps. In consequence, backward data flow is forbidden in our approach, and a sequential execution of the substeps must be able to compute all values.

3.3.2 Scheduling parallel threads

Several refined clocks can also be declared in parallel threads so they are unrelated to each other (substeps of one thread are not visible to the other thread). Thus, there is no stepwise synchronization for these clocks. Instead, synchronization is only given by steps of a higher clock which is declared outside the parallel statement and visible to both threads. In the rest of this section, we first look at a simpler situation, where we have two threads and a refined clock in addition to the module clock.

```
clock(C1) {
 l1: pause;
 $\mathcal{A}_1$
 l2: pause(C1);
 $\mathcal{A}_2$
 l3: pause;
} || {
 l4: pause;
 $\mathcal{A}_3$
 l5: pause;
}
```

In the first reaction, both threads are entered and the control-flow stops at labels l1 and l4. The first thread starts the next step of clock C0 at l1, executes the actions $\mathcal{A}_1$ and $\mathcal{A}_2$ in two substeps, and ends at l3. In the second thread, this step of clock C0 starts at label 4, includes the actions $\mathcal{A}_3$ and ends at label l5. Due to the synchronous model, the steps of both threads are executed synchronously. However, the step of clock C0 of the first thread is divided into two steps of the lower clock C1 (the first one executes the actions $\mathcal{A}_1$, and the second one executes the actions $\mathcal{A}_2$). Assume that we have the following dependencies between actions:

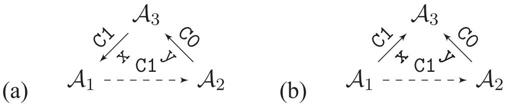


The sequential dependency between $\mathcal{A}_1$ and $\mathcal{A}_2$ is given by the source code. However, both can use the same variables of clock C1, which generally have different values in different substeps. $\mathcal{A}_3$ writes a variable that is read by $\mathcal{A}_1$, and $\mathcal{A}_2$ writes a variable that is used by $\mathcal{A}_3$. This is not necessarily a cycle because the variables imposing the dependencies can occur in different actions in $\mathcal{A}_3$. The model itself just means that all actions $\mathcal{A}_3$ are executed until label l5 is reached. Thus, splitting $\mathcal{A}_3$ into parts seems to be possible where the actions with dependencies to $\mathcal{A}_1$ are executed together with $\mathcal{A}_1$ and the other actions are executed with $\mathcal{A}_2$.

Consider a second example, which looks very similar at first glance:

```
clock(C1) {
{
l1: pause;
$\mathcal{A}_1$
l2: pause(C1);
$\mathcal{A}_2$
l3: pause;
} || {
l4: pause;
$\mathcal{A}_3$
l5: pause;
}
}
```

The code is mostly the same as the previous one—only the clock C1 is declared outside the parallel statement so that it is visible in both threads (as well as variables declared on this clock). Thus, dependencies between $\mathcal{A}_3$ and $\mathcal{A}_1$ are now also possible on clock C1. Assume the following dependencies:



The sequential dependency between $\mathcal{A}_1$ and $\mathcal{A}_2$ is still present. First, consider case (a), where a dependency by variable x of clock C1 exists from $\mathcal{A}_3$ to $\mathcal{A}_1$. The second dependency is imposed by variable y of clock C0 from $\mathcal{A}_2$ to $\mathcal{A}_3$. If the variable x can be computed without the knowledge of variable y, it is still possible to split $\mathcal{A}_3$ and execute one part with $\mathcal{A}_1$ and the other one with $\mathcal{A}_2$. However, if y is needed to determine the value of x, a cycle is present and the execution is not possible. Now, consider case (b) where the dependency goes from $\mathcal{A}_1$ to $\mathcal{A}_3$. Again, if x and y are not needed in the same actions, a split is possible. However, if both occur in the same guarded actions, the situation becomes more complicated. This action needs to be executed when the value of y is known, i.e., when $\mathcal{A}_2$ is executed. However, the value of y determined by $\mathcal{A}_1$ has to be used which is the value from the last substep. Thus, the value has to be stored so that the action in $\mathcal{A}_3$ can be executed later.

Finally, consider the following example:

```
clock(C1) {
  l1: pause;
  $\mathcal{A}_1$
  l2: pause(C1);
  $\mathcal{A}_2$
  l3: pause;
} || {
  l4: pause;
  $\mathcal{A}_3$
  l5: pause(C1);
  $\mathcal{A}_4$
  l6: pause(C1);
  $\mathcal{A}_4$
  l7: pause;
}
```

The step in the first thread is now divided into two substeps of clock **C1** and the step in the second thread into three substeps. Due to the parallel threads, the substeps are also executed in parallel. Thus, a synchronization takes place at labels **l2** and **l5**, and the actions $\mathcal{A}_1$ and $\mathcal{A}_3$ are executed together. After this first substep, there is a very similar situation to the previous example: the first thread has one step to reach label **l3** and the second one has to execute two substeps. However, in the previous example we talked about splitting the actions of the first thread. However, this seems to be very confusing, especially to the developer to decide when actions can be moved to different substeps and when not. Therefore, we only allow the actions within an instant to be executed synchronously.

4 Formal semantics

We formally define the semantics of our language extension in the style of Plotkin's *Structural Operations Semantics (SOS)* [36,37]. This formalism has already successfully been used in the context of synchronous languages [7,38,39], and the formal semantics of single-clocked Quartz [4] already exists in this format. As the name suggests, SOS rules are defined over the structure of a given program, i.e., the AST.

In sequential programming languages, a program is executed step-by-step as given in the source code. However, due to the synchronous abstraction of time, the execution of synchronous programs must follow data dependencies, which is not necessarily the order given in the source code. Hence, we cannot use SOS rules directly, but our semantics uses two sets of SOS rules: *transition rules* and *reaction rules*.

The execution of the program is based on an environment $\mathcal{E}$, which is an assignment of values to each variable of the program. The transition rules specify an interpreter: they take an environment and a given program and execute its first step, i.e., they *transform* the program according to the environment. The computation of the actual environment (which also comprises a dynamic causality analysis) is accomplished by the second set of rules, the reaction rules. In the following, we focus on the first part, the transition rules. For the reaction rules, we refer to [40,41].

4.1 Basic definitions

This section introduces some basic notations and formalizations. First, we define the basis of our temporal model, namely clocks, and their refinements. As all refinements always refine existing clocks, they can be organized in a tree-like relation, which is defined as follows.

Definition 1. (Clocks) *We write $c_1 \succ c_2$ if the clock c_2 is declared in the scope of c_1, i.e., c_1 is on a higher level* *(slower) than c_2. The relations $\succeq, \prec, \preceq$ are used accordingly. If two clocks c_1 and c_2 are independent, i.e., neither $c_1 \succeq c_2$ nor $c_1 \preceq c_2$ holds, we write c_1 # c_2.*

Two clocks are independent, i.e., c_1 # c_2, if they are either declared in parallel threads, or they are declared in two distinct parts of a sequence or an **`if`** statement. For example, the clock relations `C5` $\prec$ `C1` and `C2 # C3` hold for the program in Figure 5. In addition to the clocks, each program uses a finite set of variables and each variable is declared inside the scope of a clock. The following definition takes care of the variables and their clocks:

Definition 2. (Variables) *$\mathcal{V}$ is the set of variables of a synchronous program. Each variable $x \in \mathcal{V}$ stores a value of its domain* $\mathsf{dom}(x)$*, and it is declared in the scope of a clock, which is given by* $\mathsf{clock}(x)$*. Additionally, we denote with $\mathcal{V}^{\mathsf{IN}}$, $\mathcal{V}^{\mathsf{OUT}}$, $\mathcal{V}^{\mathsf{LOC}}$ the sets of all input, output and local variables respectively. For a variable x,* $\mathsf{default}(x)$ *denotes its* default value*.*

For example, the default value of a Boolean variable is false and that of an integer variable is 0. As an example, the clock of the variable `x` in the program `GCD2` in Figure 3 is `C1` (**`clock(x)`** = `C1`). For assigning values to variables, we use the following actions:

Definition 3. (Action) *The actions in a synchronous program are assignments of one of the following forms.*

$$\begin{array}{rcll} x & = & \tau & \quad \textit{(immediate assignment)} \\ \mathtt{next}\,(x) & = & \tau & \quad \textit{(delayed assignment)} \end{array}$$

An immediate assignment assigns the value of the expression τ directly to the variable x. A delayed assignment evaluates the value of τ directly but assigns it in the next step of $\mathsf{clock}(x)$*.*

Note that a delayed assignment takes care of the clock of the variable that is assigned. In the semantics definition of synchronous programs the values of variables are determined iteratively for each step. Therefore, a notion of *not yet known* is needed for variables. This is covered by the following definition.

Definition 4. (Environment) *An environment $\mathcal{E}$ maps each variable $x \in \mathcal{V}$ to a value of* $\mathsf{dom}(x) \cup \{\bot\}$*. Hence, the* extended domain *of a variable x additionally contains the value $\bot$, which is interpreted as* not known*. We write $\mathcal{E}\,(x)$ to retrieve the value of x in environment $\mathcal{E}$, and similarly $[\![\tau]\!]_{\mathcal{E}}$ to evaluate the expression τ with respect to the values of the variables in environment $\mathcal{E}$. The environment which is undefined for each variable is denoted with $\mathcal{E}^{\bot}$.*

In addition, we define operations on environments.

Definition 5. (Environment Combination) *For two environments $\mathcal{E}_1$ and $\mathcal{E}_2$, we define the* intersection *and* union *as follows:*

$$(\mathcal{E}_1 \sqcap \mathcal{E}_2)\,(x) := \begin{cases} v & \textit{if } v = \mathcal{E}_1\,(x) = \mathcal{E}_2\,(x) \\ \bot & \textit{otherwise} \end{cases}$$

$$(\mathcal{E}_1 \mathbin{\dot\sqcup} \mathcal{E}_2)\,(x) := \begin{cases} \mathcal{E}_1\,(x) & \textit{if } \mathcal{E}_2\,(x) = \bot \\ \mathcal{E}_2\,(x) & \textit{if } \mathcal{E}_1\,(x) = \bot \\ v & \textit{if } v = \mathcal{E}_1\,(x) = \mathcal{E}_2\,(x) \end{cases}$$

The union is only allowed if there are no conflicting values for the same variable in both environments.

Definition 6. (Environment Restriction) *A restriction of an environment $\mathcal{E}$ with respect to $\odot c$ (where $\odot \in \{\succ, \succeq, \prec, \preceq, \nsucc, \nsucceq, \nprec, \npreceq\}$) is defined as follows:*

$$(\mathcal{E})_{/_{\odot}}\, c\,(x) := \begin{cases} \mathcal{E}(x) & \textit{if } \mathsf{clock}\,(x) \odot c \\ \bot & \textit{otherwise} \end{cases}$$

Thus, $(\mathcal{E})_{/_{\odot\npreceq}}\, c$ describes the environment where all variables with a clock lower or equal to c are set to $\bot$, the values of all other variables in $\mathcal{E}$ are kept.

Definition 7. (Partial Order of Environments) *An environment $\mathcal{E}_1$ is smaller than environment $\mathcal{E}_2$ (greater resp.), if the following holds:*

$$\mathcal{E}_1 \sqsubseteq \mathcal{E}_2 :\Leftrightarrow \forall x \in \mathcal{V}.\, \mathcal{E}_1\,(x) \neq \bot \rightarrow \mathcal{E}_1\,(x) = \mathcal{E}_2\,(x)$$

Thus, at least the variables which are defined by $\mathcal{E}_1$ are defined by $\mathcal{E}_2$ with the same values.

4.2 Transition rules

The transition rules define the execution of a single step on the source code based on an existing environment for this step. Previous sections discussed the view at the model and emphasized the characteristics of the **`pause`**`(`C`)` statement as *wait for clock C*. In the transition rules, this view is pointed out by *renaming* **`pause`**`(`C`)` to **`await clock`**`(`C`)`. Analogous to the original await statement, the transition rules also use the statement **`immediate await clock`**`(`C`)` to define the behavior. Transition rules have the form

$$\langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S} \rangle \overset{\mathcal{C}}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle$$

and describe how the statement $\mathcal{S}$ is transformed to the residual statement $\mathcal{S}'$ when an instant of clock c is executed with the environment $\mathcal{E}$. Thereby, the set $\mathcal{A}$ contains the assignments which are executed during this step and the set $\mathcal{C}$ contains the clocks for which a corresponding **`pause`** statement is reached during the execution. Thus, $\mathcal{C}$ collects the clocks which can be used for the next step. The statement clock $C_{\mathcal{S}}$ is the lowest clock the statement is defined in.

In the following, we only give the rules for the new statements of our extension. All the other rules are similar to the original definition for single-clocked Quartz, and they can be found in Appendix 1. Additional details about their definition can be found in [35].

Now consider a simple example for the transition rules. Thereby, depending on the input **i**, the following statement $\mathcal{S}$ can either be derived to $\mathcal{S}_1'$ or $\mathcal{S}_2'$:

```
        x = true;
        if(i) {                   x = false;
        y = true;             S1': l2: pause(C0);
        l1: pause(C0);            l3: pause(C0);
    S:  x = false;                y = false;
        l2: pause(C0);
        }
        l3: pause(C0);        S2': y = false;
        y = false;
```

For an instant where input `i` holds, i.e., $\mathcal{E}_1(\mathbf{i}) = \mathsf{true}$, the **if**-branch is entered and the statement is reduced by the transition rules to:

$$\langle \mathcal{E}_1, \mathtt{C0}, \mathcal{S}\rangle \overset{\mathcal{C}}{\twoheadrightarrow} \langle \mathcal{S}_1', \{\mathtt{x}{=}\mathbf{true}, \mathtt{y}{=}\mathbf{true}\}, \{\mathtt{C0}\}\rangle$$

For an instant where $\mathcal{E}_2(\mathtt{i}) = \mathsf{false}$ holds, the **if**-branch is not entered and the statement is transformed by the transition rules to:

$$\langle \mathcal{E}_2, \mathtt{C0}, \mathcal{S}\rangle \overset{\mathcal{C}}{\twoheadrightarrow} \langle \mathcal{S}_2', \{\mathtt{x}{=}\mathbf{true}\}, \{\mathtt{C0}\}\rangle$$

Note that the **if** statement is completely removed after it is reached. The condition is only checked when the statement is reached and in this instant it is substituted with the one or the other branch depending on the evaluation of the condition.

The transition rules which are used to define the semantics of single-clocked Quartz in [4] use a Boolean flag instead of the set $\mathcal{C}$. In the single clock case it is sufficient to indicate whether a **pause** statement is reached and whether the macro step terminated. For refined clocks, we collect the clocks of all **pause** statements which are reached to be able to determine a clock for the next step. However, the same information is still available by checking the emptiness of $\mathcal{C}$ as it can be found in the rules.

Exemplarily, the rules for the new **pause**(C) statement, or as it is called in the transition rules **await clock**(C), are explained. When this statement is reached, it is changed to **immediate** → **await clock**(C). More important, the clock C is added to the set $\mathcal{C}$ which indicates that a new step of clock C can be done. The rules for **immediate await clock**(C) only proceed with the execution when a step on the associated clock is performed.

The rules for the clock declaration (c_1 and c_2) are straightforward. Both rules update the statement clock to the current declaration. The rules differ in whether the local block is executed in an instant or not. If this is the case, the whole block is removed, otherwise it remains with the residual statement.

4.3 Program execution

Based on the transition rules, the execution of a program can be defined as a sequence of tuples.

$$\begin{gathered}\left(\mathcal{E}_0^{\mathsf{PRV}}, \mathcal{E}_0^{\mathsf{CUR}}, \mathcal{E}_0^{\mathsf{NXT}}, \mathcal{E}_0^{\mathsf{ASS}}, \mathcal{S}_0, c_0\right),\\ \left(\mathcal{E}_1^{\mathsf{PRV}}, \mathcal{E}_1^{\mathsf{CUR}}, \mathcal{E}_1^{\mathsf{NXT}}, \mathcal{E}_1^{\mathsf{ASS}}, \mathcal{S}_1, c_1\right),\\ \ldots,\\ \left(\mathcal{E}_n^{\mathsf{PRV}}, \mathcal{E}_n^{\mathsf{CUR}}, \mathcal{E}_n^{\mathsf{NXT}}, \mathcal{E}_n^{\mathsf{ASS}}, \mathcal{S}_0, c_n\right)\end{gathered}$$

Thereby, each tuple coincides with an instant in which c_i holds. The environments $\mathcal{E}_i^{\mathsf{CUR}}$ store the current values of the variables, $\mathcal{E}_i^{\mathsf{PRV}}$ hold the values of the values in the previous step (w. r. t. the clock c_i), and $\mathcal{E}_i^{\mathsf{NXT}}$ hold the values of delayed assignments which have to be committed in the next step (w. r. t. the clock c_i). In addition, the environments $\mathcal{E}_i^{\mathsf{ASS}}$ hold the values of the current step which has been already assigned by an immediate assignment. The module is initially started with the module clock, and thus, $c_0 = \mathbf{C0}$ and $\mathcal{S}_0$ is the whole program. All other instants $(0 \leq i < n)$ are defined by the transition rules:

$$\langle \mathcal{E}_i, \mathtt{C0}, \mathcal{S}_i\rangle \overset{\mathcal{C}_i}{\twoheadrightarrow} \langle \mathcal{S}_{i+1}, \mathcal{A}, \mathcal{C}\rangle$$

Thereby, the clock for the following instant is defined by the **pause** statements which are reached:

$$c_{i+1} \in \mathcal{C}, \nexists c \in \mathcal{C}.\ c \succ c_{i+1}$$

With the clock c_i, a new step of this clock is started. Thus, the environment $\mathcal{E}_i^{\mathsf{PRV}}$ which holds the values of each variable from its last step has to be updated for the variables with clock c_i or lower clocks. The values of all other variables are retained:

$$\mathcal{E}_i^{\mathsf{PRV}} \quad = \quad \left(\mathcal{E}_{i-1}^{\mathsf{PRV}}\right)_{/_{c_i \npreceq}} \dot{\sqcup} \left(\mathcal{E}_{i-1}^{\mathsf{CUR}}\right)_{/_{c_i \preceq}}$$

The same holds for the environment $\mathcal{E}_i^{\mathsf{CUR}}$. However, here it is only required that the values of the variables with a clock not lower or equal to c_i are kept and all variables are assigned a value:

$$\mathcal{E}_i^{\mathsf{CUR}} \sqsupseteq \left(\mathcal{E}_{i-1}^{\mathsf{CUR}}\right)_{/_{c_i \not\preceq}}$$
$$\nexists x \in \mathcal{V}.\, \mathcal{E}_i^{\mathsf{CUR}}(x) = \bot$$

The definition of the both environments $\mathcal{E}_i^{\mathsf{NOWASS}}$ and $\mathcal{E}_i^{\mathsf{NXTASS}}$ are used to treat the executed actions in environments. Thus, if there is an action in $\mathcal{A}$ which sets the variable x to τ, the value of τ evaluated by $\mathcal{E}_i^{\mathsf{CUR}}$ is assigned by $\mathcal{E}_i^{\mathsf{NOWASS}}$. Accordingly, $\mathcal{E}_i^{\mathsf{NXTASS}}$ holds the values of the delayed assignments:

$$\mathcal{E}_i^{\mathsf{NOWASS}}(x) := \begin{cases} [\![\tau]\!]_{\mathcal{E}_i^{\mathsf{CUR}}} & \text{if } x = \tau \in \mathcal{A} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{E}_i^{\mathsf{NXTASS}}(x) := \begin{cases} [\![\tau]\!]_{\mathcal{E}_i^{\mathsf{CUR}}} & \text{if } \mathtt{next}(x) = \tau \in \mathcal{A} \\ \bot & \text{otherwise} \end{cases}$$

The environments $\mathcal{E}_i^{\mathsf{NXT}}$ and $\mathcal{E}_i^{\mathsf{ASS}}$ are updated according to the executed assignments. A delayed assignment from the last step is also transferred to an immediate one of the new step:

$$\begin{aligned} \mathcal{E}_i^{\mathsf{NXT}} &= \left(\mathcal{E}_{i-1}^{\mathsf{NXT}}\right)_{/_{c_i \not\preceq}} \dot{\sqcup}\ \mathcal{E}_i^{\mathsf{NXTASS}} \\ \mathcal{E}_i^{\mathsf{ASS}} &= \left(\mathcal{E}_{i-1}^{\mathsf{ASS}}\right)_{/_{c_i \not\preceq}} \dot{\sqcup} \left(\mathcal{E}_{i-1}^{\mathsf{NXT}}\right)_{/_{c_i \preceq}} \dot{\sqcup}\ \mathcal{E}_i^{\mathsf{NOWASS}} \end{aligned}$$

With the clock of the next instant c_{i+1} a new step of this clock is started. It is also necessary to ensure that the variables had the correct value for the step:

$$\forall x \in \left(\mathcal{V}^{\mathsf{LOC}} \cup \mathcal{V}^{\mathsf{OUT}}\right).\mathsf{clock}(x) \preceq c_{i+1} \rightarrow$$
$$\mathcal{E}_i^{\mathsf{CUR}}(x) := \begin{cases} \mathcal{E}_i^{\mathsf{ASS}}(x) & \text{if } \mathcal{E}_i^{\mathsf{ASS}}(x) \neq \bot \\ \mathcal{E}_i^{\mathsf{PRV}}(x) & \text{if } x \text{ is memorized variable} \\ \mathsf{default}(x) & \text{if } x \text{ is event variable} \end{cases}$$

Since $0 \leq i < n$, we need to initially define $\mathcal{E}_{-1}^{\mathsf{PRV}} = \mathcal{E}_{-1}^{\mathsf{NXT}} = \mathcal{E}_{-1}^{\mathsf{ASS}} = \mathcal{E}^{\bot}$ and $\forall x \in \mathcal{V}.\, \mathcal{E}_{-1}^{\mathsf{CUR}}(x) := \mathsf{default}(x)$.

5 Compilation

This section explains the compilation of Quartz with refined clocks. Similar to traditional Quartz, we also use guarded actions as an intermediate format. However, the intermediate format has to be extended appropriately so that it can represent systems with refined clocks. The translation to the intermediate format is presented in Section 5.1. Based on this, we discuss two possible targets: hardware synthesis in Section 5.2 and software synthesis in Section 5.3.

5.1 Translation to the intermediate format

Similar to the extension of Quartz with refined clocks, we also have to extend the intermediate format. As already shown in Section 2.3, the intermediate format represents the behavior of a system by guarded actions defined over a set of explicitly declared variables. Obviously, clocks and their relations are additionally needed to describe the data flow in the context of refined clocks, and the extended intermediate format contains all this technical information.

In the single-clock case, each guarded action is bound to at least one label which defines the control flow location in the source code where the action is executed from. For refined clocks, it is now necessary to only execute the actions when (1) the label holds and (2) the according clock ticks. Therefore, the label in the guard is strengthened with the corresponding clock, which requires to introduce variables for the clocks.

Before we go into details about the compilation, we first recall the remarks from Section 3.3. There, we saw that a **`pause`** statement is only left when its clock is present (it will block until the given clock ticks). Consider the following guarded action (see also Figure 2):

$$\mathtt{l2} \wedge \mathtt{i1} > 4 \Rightarrow \mathtt{o1} = \mathtt{i1}$$

This action originates from a single clock example but it can be also extended by the module clock `C0` (which is the clock of all labels in the single clock case):

$$(\mathtt{l2} \wedge \mathtt{C0}) \wedge \mathtt{i1} > 4 \Rightarrow \mathtt{o1} = \mathtt{i1}$$

In the single-clock case, both guarded actions do not make any difference since `C0` holds in every instant. However, we will extend the idea to refined clocks, where clocks do not generally tick in every instant, and store for each variable its clock. In particular, the Boolean control flow labels are also bound to a clock. Finally, without going into technical details, since the intermediate format stores everything which is needed for further processing, also the dependencies between clocks, i.e., the clock tree of the system, are stored. A concrete example of a clock tree was given in Figure 5, which would in this case be contained in the intermediate format.

The compilation algorithm for refined clocks to the extended intermediate format can be found in Appendix 2. It is based on the original compilation algorithm [4]. In particular, we also use the notion of *surface* and *depth*. Thereby, the surface of a statement are the actions which can be executed in the first instant, and the depth are the actions which can be executed in the following instants.

The algorithm basically works in the same way as the original one, it traverses the AST of the programm and determines with `CompileSurface` the actions which are executed in the instant starting from the current position. The entry point is defined by the function `Compile` which calls `CompileSurface` and `CompileDepth` for the whole program. It also sets the module clock `C0` as the highest one, because it is not explicitly defined. The labels of the `pause` statements are strengthened by the clock as described above. In addition, the abort and suspend conditions need only to be checked on the corresponding labels. On labels which are defined on a lower clock inside of an abort block, those conditions do not need to be checked. Therefore, the algorithm uses maps to store the conditions of the surrounding abort and suspend blocks related to each clock. At a `pause` statement, the condition for the clock just needs to be added. For a detailed description of the compilation algorithm consider [40,42]. The guarded actions of the example program `GCD2` are given in Figure 7.

5.2 Hardware synthesis

The next step of the translation to hardware circuits is an equation system. Given a library for all operators in our data flow, the equations can syntactically be translated to any hardware description language such as verilog or VHDL. In principle, the translation can be also used to generate software but, as shown in Section 5.3, there is a more efficient translation for that purpose.

In the equation system, we use three different kinds of equations. The first type represents *wires* which are directly connected to some logic so that the computed value is immediately available. Such an immediate equation has the following form:

$$x = \tau$$

for a variable x set to the current value of τ.

State elements such as *registers* are represented by the remaining equations. Each one is defined by two equations, one for the initial step and one for subsequent transitions.

$$\begin{array}{rcl} \mathtt{init}(x) & = & \tau_1 \\ \mathtt{next}(x) & = & \tau_2 \end{array}$$

$$\begin{array}{rrcl}
\mathtt{C0} \wedge \mathsf{st} \Rightarrow & \mathtt{x} & = & \mathtt{a} \\
\mathtt{C0} \wedge \mathsf{st} \Rightarrow & \mathtt{y} & = & \mathtt{b} \\
\mathtt{C0} \wedge \mathsf{st} \wedge \mathtt{x} > 0 \wedge \mathtt{x} \geq \mathtt{y} \Rightarrow & \mathtt{next}\,(\mathtt{x}) & = & \mathtt{x} - \mathtt{y} \\
\mathtt{C0} \wedge \mathsf{st} \wedge \mathtt{x} > 0 \wedge \mathtt{x} \not\geq \mathtt{y} \Rightarrow & \mathtt{next}\,(\mathtt{x}) & = & \mathtt{y} - \mathtt{x} \\
\mathtt{C1} \wedge \mathtt{l} \wedge \mathtt{x} > 0 \wedge \mathtt{x} \geq \mathtt{y} \Rightarrow & \mathtt{next}\,(\mathtt{x}) & = & \mathtt{x} - \mathtt{y} \\
\mathtt{C1} \wedge \mathtt{l} \wedge \mathtt{x} > 0 \wedge \mathtt{x} \not\geq \mathtt{y} \Rightarrow & \mathtt{next}\,(\mathtt{x}) & = & \mathtt{y} - \mathtt{x} \\
\mathtt{C1} \wedge \mathtt{l} \wedge \mathtt{x} \not> 0 \Rightarrow & \mathtt{gcd} & = & \mathtt{y}
\end{array}$$

Data-Flow Actions

$$\begin{array}{rrcl}
\mathtt{C0} \wedge \mathsf{st} \Rightarrow & \mathtt{next}\,(\mathtt{l}) & = & \mathsf{true} \\
\mathtt{C1} \wedge \mathtt{l} \wedge \mathtt{x} > 0 \Rightarrow & \mathtt{next}\,(\mathtt{l}) & = & \mathsf{true}
\end{array}$$

Control-Flow Actions

Figure 7 Guarded actions of program `GCD2`.

Note that the *clock* which defines the steps is now the hardware clock, which is generally different to the logical clocks of the source language.

5.2.1 *Control flow*

For the translation of the control flow, every label is considered separately. Such a label ℓ can be written by multiple delayed guarded actions (note that the control flow does not contain immediate actions). Assume that the label ℓ is written by the following actions:

$$\begin{array}{l}
\gamma_1 \Rightarrow \mathtt{next}(\ell) = \mathsf{true} \\
\gamma_2 \Rightarrow \mathtt{next}(\ell) = \mathsf{true} \\
\qquad\qquad \ldots \\
\gamma_n \Rightarrow \mathtt{next}(\ell) = \mathsf{true}
\end{array}$$

The label can be set by this guarded actions, and it remains active until its clock holds. Therefore, the actions are combined to define a register in the following way:

$$\begin{array}{rcl}
\mathtt{init}\,(\ell) & = & \mathsf{false} \\
\mathtt{next}\,(\ell) & = & \underbrace{\gamma_1 \vee \gamma_2 \vee \ldots \vee \gamma_3}_{\text{guards}} \vee \underbrace{(\ell \wedge \neg \mathsf{clock}(\ell))}_{\text{default}}
\end{array}$$

The expression to set the register is split into two parts. The first one is given by the guards of the control flow guarded actions which ensures that the label is set when one of the guards hold. The second part is the default value which ensures that the label remains activated as long as no tick of its clock occurs. The special start label st is translated as follows:

$$\begin{array}{rcl}
\mathtt{init}\,(\mathsf{st}) & = & \mathsf{true} \\
\mathtt{next}\,(\mathsf{st}) & = & \mathsf{st} \wedge \neg \mathsf{clock}\,(\mathsf{st})
\end{array}$$

Thus, we just set it initially, and reset it for the rest of the execution.

5.2.2 *Data flow*

The translation of the data flow is more sophisticated since we have to consider the following issues: (1) data flow variables can be written by delayed *and* immediate assignments and (2) delayed assignments do not necessarily take place in the next instant, instead the value has to be kept until the next tick of the variables clock. In general, two registers are needed for each variable (for some special cases, the following general solution can be optimized but we will present the full solution for the sake of completeness). Assume that a variable x is written by the following guarded actions:

$$\begin{array}{ll}
\gamma_1^i \Rightarrow x = \tau_1^i & \gamma_1^d \Rightarrow \mathtt{Next}(x) = \tau_1^d \\
\gamma_2^i \Rightarrow x = \tau_2^i & \gamma_2^d \Rightarrow \mathtt{Next}(x) = \tau_2^d \\
\cdots & \cdots \\
\gamma_n^i \Rightarrow x = \tau_n^i & \gamma_m^d \Rightarrow \mathtt{Next}(x) = \tau_m^d
\end{array}$$

For a variable x, two new variables are introduced which are converted to a register:

- $x^{\mathtt{nxt}}$
  Delayed assignments are expected to take place at the next occurrence of the clock of x. The variable $x^{\mathtt{nxt}}$ stores values from those delayed assignments, until the clock holds.
- $x^{\mathtt{prv}}$
  Since the steps of a certain clock do no longer coincide with the instants, the values of variables have to be kept for the whole step. Therefore, the variable $x^{\mathtt{prv}}$ stores the value of x from the previous instant.

The equations can then be defined as follows:

$$\begin{array}{rcl}
\mathtt{init}\left(x^{\mathtt{nxt}}\right) & = & \mathsf{default}(x) \\
\mathtt{next}\left(x^{\mathtt{nxt}}\right) & = & \begin{cases} \tau_1^d & : \gamma_1^d \\ \tau_2^d & : \gamma_2^d \\ \vdots & \vdots \\ \tau_m^d & : \gamma_m^d \\ \mathtt{trans}\,(x) & : \text{default} \end{cases} \\
\mathtt{init}\left(x^{\mathtt{prv}}\right) & = & \mathsf{default}(x) \\
\mathtt{next}\left(x^{\mathtt{prv}}\right) & = & x \\
x & = & \begin{cases} \tau_1^i & : \gamma_1^i \\ \tau_2^i & : \gamma_2^i \\ \vdots & \vdots \\ \tau_m^i & : \gamma_n^i \\ x^{\mathtt{nxt}} & : \mathsf{clock}(x) \\ x^{\mathtt{prv}} & : \text{default} \end{cases}
\end{array}$$

where the expression $\mathtt{trans}\,(x)$ depends on the storage type of the variable x:

$$\mathtt{trans}\,(x) := \begin{cases} \mathsf{default}(x) & : x \text{ is event variable} \\ x & : x \text{ is memorized variable} \end{cases}$$

Optimizations are possible e. g. if no delayed assignments exists and $\mathtt{trans}\,(x)$ is a constant value (e. g. false). In this case, the variable $x^{\mathtt{nxt}}$ can completely be removed because it always holds a constant value. Similarly, optimizations are possible if there are not immediate assignments for a variable x. In this case, the translation shown for the control flow can be used.

5.2.3 Scheduling

In addition to the translation of the control flow and the data flow, we have to consider the clocks for the synthesis. In single-clock Quartz, this is simple since the hardware clock coincides with the module clock of the Quartz module. Thus, in each clock cycle one instant of the module is executed. The hardware synthesis for refined clocks is based on the same idea but for independent clocks the one or the other instant can be executed. In addition, not every clock is allowed to occur in every instant due to the restrictions imposed by the clock tree and the control flow. Restrictions can also be imposed by the data flow, if one thread waits for a value which is computed by an independent step in a later instant. To sum up, scheduling the clocks according to the semantics requires some analysis.

In the following, we assume that communication between unrelated clocks is only done by delayed actions (for a generalization see [40,41]). This means that no data dependencies exist for unrelated clocks and communication among them are synchronized by a common higher clock. In this case, there is no need to consider data-flow dependencies and we can describe a scheduler for the clocks only by the control flow. We call a clock C *enabled* if one of its labels holds:

$$\mathsf{enabled}(C) := \bigvee_{\ell \in \mathcal{L}, \mathsf{clock}(\ell) = C} \ell$$

A clock is only allowed to tick, if at least one of the related **pause** statements are reached. In addition, it can only tick if no lower clock is enabled, because execution should synchronize on common **pause** statements. Therefore, we also define the sets of all lower and all higher clocks of C by:

$$\begin{array}{rcl}
\mathsf{lower}(\mathcal{C}) & := & \{c \in \mathcal{C} \mid c \prec C\} \\
\mathsf{higher}(\mathcal{C}) & := & \{c \in \mathcal{C} \mid c \succ C\}
\end{array}$$

With these definitions, we can finally construct the equation for the clock C as follows:

$$C = \underbrace{\mathsf{enabled}(C) \wedge \bigwedge_{c \in \mathsf{lower}(\mathcal{C})} \neg \mathsf{enabled}(c) \vee}_{\text{tick by its own}} \underbrace{\bigvee_{c \in \mathsf{higher}(\mathcal{C})} c}_{\text{tick forced by higher clock}}$$

Thus, a clock can tick by its own, if it is enabled, but no lower clock is. In addition, a clock tick can be forced by a higher clock which also includes all lower ones. This is to trigger the delayed assignments also for the lower clocks.

5.3 Software synthesis

Synchronous languages can be used to build hardware and software from the same description. One possible solution for this is a software synthesis which *simulates* the hardware that is described above. However, there are more efficient solutions. One possibility is based on the extended finite state machine. Thereby, the possible combinations of labels form the states. The guarded actions are grouped by the labels which occur in their guards and are assigned to the corresponding states. Thus, in each state only the guarded actions which are possibly executed have to be evaluated.

With the introduction of refined clocks, there exists another parameter to classify the guarded actions. Therefore, the guarded actions are first divided by the clocks (of the labels) which occur in their guards. The guarded actions of each clock are combined to a *task*. The advantage is that local variables of a task are the local variables of the clock and do not have to be made visible to other tasks. Inputs of a task come from the higher level and outputs go back to the higher level or to a lower one.

To complete one step, a module usually has to complete several substeps. The substeps can be associated by unrelated clocks, thus, they can be executed independently. With the model of tasks, the inputs and the according state can be send to the tasks and they can concurrently execute several substeps. The tasks can be scheduled dynamically whenever a lower clock level is *entered*. Thus, tasks model the parallelism which is inherent to refined clocks.

5.4 Example

In this section, we will discuss by an example that refined clocks can be used to relax over-synchronization and that this advantage can be used for a more liberal code generation. The software realization does not need to introduce needless synchronization, and hardware implementations can use different schedulers for the refined clocks to control the trade-off between resources (space of the hardware design) and execution time.

Consider the following example which consists of two parallel threads with two unrelated clocks. In each one, the same resource is used—for illustration, assume that it is a multiplier.

```
clock(C1) {
  ...
  l1: pause(C1);
  a = b * c;
  ...
} || clock(C2) {
  ...
  l2: pause(C2);
  x = y * z;
  ...
}
```

Without refined clocks, synchronization of both threads would be necessary (due to the semantics) on each **pause** statement. The original single-clocked hardware synthesis considers each instant as a clock cycle. Therefore, using the same multiplication unit for both multiplications would require a reachability analysis to ensure that both are not executed in the same clock cycle.

Refined clocks relax the need for synchronization and only require them for the same clocks. Therefore, synchronization is not necessary for the above example. The scheduler, as it is described in Section 5.2.3, can (1) execute both steps containing multiplications together or it can (2) ensure that both steps are executed one after the other. For the first case, two multipliers are necessary, and both are used in parallel. For the second case, since the scheduler ensures the mutual exclusive access, only one multiplication unit is necessary, since the multiplications are executed one after the other. To summarize, both cases differ in space (of the hardware design) and (execution) time in terms of clock cycles. From this point of view, the first case can also be achieved without refined clocks. However, refined clocks initially introduce the possibility of selecting between space and time.

6 Related work

Using more than one clock in a system is a quite common approach to deal with timing, synchronization, and independent execution in synchronous systems, even though the term *clock* can be misleading since it is used for many different concepts: a hardware developer will probably understand by clock a periodic signal whose occurrence is based on a fixed physical time. In this case, the clock signal is typically fed in from the environment (clock generator) into the actual circuit and is used to drive the execution. Another interpretation is given from synchronous dataflow languages like Lustre or Signal where each signal has a clock that identifies the *availability/presence of data*. Thereby, the presence of data can depend on the presence of other data *and* also on other data values. These clocks are not necessarily all given by the environment, and can be instead computed by the system from the given ones. Finally, the imperative languages mentioned in this article, Esterel and Quartz, are single-clock synchronous languages where a clock is used to separate the execution into single reaction steps. If these languages are translated to synchronous hardware circuits, each step *can* be mapped to a hardware-clock cycle, but there is no reason to compile it in this way. Different approaches related to refined clocks are introduced and compared to the presented work in the following.

6.1 Esterel

The synchronous language Esterel is quite similar to the language Quartz described in Section 2. A difference which should be pointed out is the interpretation of the terms *signal* and *variable*. Since both are used synonymously in Quartz, Esterel makes a clear distinction between both. Where the Esterel signals behave like Quartz signals/variables, which are only allowed to have one value per step, Esterel variables can be assigned multiple times. When a variable is read, its last assigned value is used:

```
X := 0;
emit S1(X)
X := X + 1;
emit S1(X);
```

The example is taken from [43]. The variable X is assigned two times and used to set the values of the signals S1 and S2. Thereby, S1 receives the value 0 and S1 receives the value 1. Even though these variables of Esterel are useful, they also have some limitations: It is not allowed to write and read them in independent threads. Also, since Esterel forbids instantaneous loops, the introductory GCD2 example cannot be converted to Esterel. Moreover, the Esterel variables only provide one simplified abstraction layer for data. In contrast, refined clocks can be *arbitrarily* deep nested, can be used in parallel threads, and they can also interact with preemption statements.

6.2 Multiclock Esterel

Originally, Esterel also has the single-clock abstraction of steps, but in the past, it has been enriched with two different multiclock extensions. They are both named *multiclock* Esterel and introduced in [44,45].

Berry and Sentovich [44] introduced their version of multiclock Esterel. Their work addresses the need to design systems with multiple clock domains in a modular way. Each module can run on its own clock, where each step of the module coincides with a clock tick of the module's clock. The modules itself are still single-clock modules with the possibility to call other modules on a different clock. To communicate data between clock domains, the authors defined two possible communication devices, named *sampler* and *reclocker*. Finally, a system consists of different modules each running at their own clock and communicating by the defined communication devices. In case of this version of multiclock Esterel, the clocks trigger the computation of each module and have to be additionally provided by the outer environment, which can be, e.g., a hardware clock.

The second multiclock Esterel extension was proposed by Rajan and Shyamasundar [45,46]. Their solution introduces a new statement which allows to *override* the clock locally by an expression based on known signals. The local statement tick is then based on this new clock expression. Finally, the signals where the local clocks are defined with, have also to be provided by the outer environment. The difference to Berry's extension is that no dedicated clock signal is used, but any signal can be used to define a new tick.

Both extensions basically allow to define new (arbitrary) clocks for a module or a code block. However, they do not allow to access multiple clocks at the same time. In addition, the clocks have a different meaning here, since they are intended to be given from the environment to trigger computations. Instead, our Quartz extension refines the inner descriptions where clocks are used to divide steps into substeps. This is only used for modeling, not for execution.

6.3 Lustre and Signal

Multi-clocked systems can also be described by the synchronous language Lustre [3,47]. Each Lustre program basically consists of a set of equations over data streams. In addition to functions and delays, there are two operators to change the rate/clock of a stream. The clock of a stream identifies the positions where a value is present. The *downsampling operator* **when** takes a stream of arbitrary type and a Boolean stream and keeps only the events of the first one at those instants where the second one is true. The upsampling operator **current** undoes a previous downsampling operation by inserting the last known value in the missing locations of the stream. Each node has a so-called base clock and at least one input of the node must run on this clock. New signal definitions always come with the definition of the clock. Hence, since upsampling only undoes the last downsampling, there is no means to refine the base clock. Therefore, the base clock is the *fastest* clock of a node and contains all instants at which any computation or communication may happen. Lustre specifications are completely deterministic due to their bottom-up design from the base clock.

In contrast to this, the polychronous language Signal [48-50] is also based on multiple clocks. While the syntax looks almost like Lustre, its semantics is very different due to its assumption that there may not be a base clock. As a consequence, Signal specifications are relational and not functional like Lustre: they do not describe a single behavior, but several possible ones, which differ in the clocks. Hence, Signal solves most of the problems mentioned in introduction—however, the price one has to pay for this powerful model is that input/output determinism is generally lost. It can be guaranteed if the program is shown to be endochronous [51] or weakly endochronous [52]. While endochrony proves determinism by the existence of a base clock (usually called master trigger in this context), weak endochrony also reveals some internal nondeterminism that can safely be exploited for a more efficient execution. Unfortunately, weak endochrony cannot be automatically checked in general. However, Signal cannot solve all the problems we have mentioned in the introduction. In particular, the definition of program functions which hide a sequential computation in an instantaneous expression is not possible. For example, a basic Signal node which instantaneously computes the GCD (i.e., its result is available at the same instant when the inputs arrive), cannot be replaced by other nodes running at a higher rate (cf. the introductory `GCD2` example).

Both Lustre and Signal deal with inputs and outputs based on different clocks. Again, those clocks are different to the clocks of the Quartz extension since their computation steps are refined internally. In addition, since Signal is able to solve some of the introductory problems, it is a matter of taste whether to use a (descriptive) data flow language like Signal or to use a control-flow-based language like Quartz.

6.4 Discrete event

The discrete event languages Verilog and VHDL are hardware description languages used to model circuits. A *simulation* semantics is defined for them which allows similar to the Esterel variables multiple updates to signals in so-called delta cycles where the physical simulation time does not proceed. In this way, a signal can have multiple values in a clock cycle. However, besides the fact, that this behavior is hard to survey, it is only available for hardware simulation, hence not for synthesis and also not for software designs.

Finally, the same observation as for Esterel holds, i.e., the delta-cycle changes only provide a single abstraction for data and cannot influence the control-flow of substeps. Even more, Esterel and Quartz provides more rich control-flow statements which can be used for the whole design process including code generation. With refined clocks, the same rich control-flow statements can be used for arbitrarily many abstraction layers.

7 Conclusion

Imperative synchronous languages are limited so far to a single clock abstraction of time which imposes restrictions to the programmer. We introduce refined clocks as a language extension to the language Quartz. This article presents the problems introduced by this new extension and it shows how they can be solved in a practical way. It formally defines the semantics for the new extension and a compilation algorithm to translate the programs to a new intermediate format. In addition, synthesis to hardware and to software is presented. It is also shown how these synthesis procedures can benefit from the new features which have been introduced.

Appendix 1: Transition rules

The transition rules defining the semantics of the language extesnion are given in Figure 8 for the basic statements, in Figure 9 for the parallel execution, in Figure 10 for strong abortion, and in Figure 11 for the strong suspension.

A Transition Rules

Assignments

$$(a_1)\qquad \frac{\mathcal{E}(\mathtt{x}) = [\![\tau]\!]_{\mathcal{E}}}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathtt{x} = \tau\rangle \overset{c}{\twoheadrightarrow} \langle \mathbf{nothing}, \{\mathtt{x} = \tau\}, \{\}\rangle}$$

$$(a_2)\qquad \langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{next}(\mathtt{x}) = \tau\rangle \overset{c}{\twoheadrightarrow} \langle \mathbf{nothing}, \{\mathbf{next}(\mathtt{x}) = \tau\}, \{\}\rangle$$

Time Consuming Statements

$$(p_1)\qquad \langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{await\ clock}(C)\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\mathbf{immediate}\\ \mathbf{await\ clock}(C)\end{array}, \{\}, \{C\}\right\rangle$$

$$(p_2)\qquad \frac{c = C}{\left\langle \mathcal{E}, C_{\mathcal{S}}, \begin{array}{l}\mathbf{immediate}\\ \mathbf{await\ clock}(C)\end{array}\right\rangle \overset{c}{\twoheadrightarrow} \langle \mathbf{nothing}, \{\}, \{\}\rangle}$$

$$(p_3)\qquad \frac{c \neq C}{\left\langle \mathcal{E}, C_{\mathcal{S}}, \begin{array}{l}\mathbf{immediate}\\ \mathbf{await\ clock}(C)\end{array}\right\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\mathbf{immediate}\\ \mathbf{await\ clock}(C)\end{array}, \{\}, \{C\}\right\rangle}$$

Clock Definitions

$$(c_1)\qquad \frac{\mathcal{C} \neq \{\} \ \wedge\ \langle \mathcal{E}, C, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{clock}(C)\{\ \mathcal{S}\ \}\rangle \overset{c}{\twoheadrightarrow} \langle \mathbf{clock}(C)\{\ \mathcal{S}'\ \}, \mathcal{A}, \mathcal{C}\rangle}$$

$$(c_2)\qquad \frac{\mathcal{C} = \{\} \ \wedge\ \langle \mathcal{E}, C, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{clock}(C)\{\ \mathcal{S}\ \}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}$$

Conditional Statements

$$(i_1)\qquad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \ \wedge\ \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1', \mathcal{A}_1, \mathcal{C}_1\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{if}(\sigma)\ \mathcal{S}_1\ \mathbf{else}\ \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1', \mathcal{A}_1, \mathcal{C}_1\rangle}$$

$$(i_2)\qquad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \ \wedge\ \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_2', \mathcal{A}_2, \mathcal{C}_2\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathbf{if}(\sigma)\ \mathcal{S}_1\ \mathbf{else}\ \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_2', \mathcal{A}_2, \mathcal{C}_2\rangle}$$

Sequence

$$(s_1)\qquad \frac{\mathcal{C}_1 \neq \{\} \ \wedge\ \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1', \mathcal{A}_1, \mathcal{C}_1\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1 ;\ \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1' ;\ \mathcal{S}_2, \mathcal{A}_1, \mathcal{C}_1\rangle}$$

$$(s_2)\qquad \frac{\mathcal{C}_1 = \{\} \ \wedge\ \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1', \mathcal{A}_1, \mathcal{C}_1\rangle \ \wedge\ \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_2', \mathcal{A}_2, \mathcal{C}_2\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1 ;\ \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_2', \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_2\rangle}$$

Figure 8 **Transition rules I (basic statements).**

Parallel Threads

$$(p_1)\quad \frac{\langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1', \mathcal{A}_1, \mathcal{C}_1\rangle \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_2', \mathcal{A}_2, \mathcal{C}_2\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}_1 \;||\; \mathcal{S}_2\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}_1' \;||\; \mathcal{S}_2', \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2\rangle}$$

Figure 9 **Transition rules II (parallel execution).**

(Strong) Abortion

$$(a_1)\quad \frac{\mathcal{C} = \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{abort}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{nothing}, \mathcal{A}, \mathcal{C}\rangle}$$

$$(a_2)\quad \frac{\forall c \in \mathcal{C}.c \prec C_{\mathcal{S}} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{abort}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{abort}\ \mathcal{S}'\ \texttt{when}(\sigma), \mathcal{A}, \mathcal{C}\rangle}$$

$$(a_2)\quad \frac{\forall c \in \mathcal{C}.c \succeq C_{\mathcal{S}} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{abort}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{immediate abort}\ \mathcal{S}'\ \texttt{when}(\sigma), \mathcal{A}, \mathcal{C}\rangle}$$

$$(a_3)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{immediate abort}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{nothing}, \{\}, \{\}\rangle}$$

$$(a_4)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\left\langle \mathcal{E}, C_{\mathcal{S}}, \begin{array}{l}\texttt{immediate abort}\\ \mathcal{S}\\ \texttt{when}(\sigma)\end{array}\right\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\texttt{immediate abort}\\ \mathcal{S}'\\ \texttt{when}(\sigma)\end{array}, \mathcal{A}, \mathcal{C}\right\rangle}$$

$$(a_5)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad\wedge\quad \mathcal{C} = \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{immediate abort}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{nothing}, \mathcal{A}, \mathcal{C}\rangle}$$

Figure 10 **Transition rules III (strong abortion).**

(Strong) Suspension

$$(a_1)\quad \frac{\mathcal{C} = \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{nothing}, \mathcal{A}, \mathcal{C}\rangle}$$

$$(a_2)\quad \frac{\forall c \in \mathcal{C}.c \prec C_{\mathcal{S}} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{suspend}\ \mathcal{S}'\ \texttt{when}(\sigma), \mathcal{A}, \mathcal{C}\rangle}$$

$$(a_2)\quad \frac{\forall c \in \mathcal{C}.c \succeq C_{\mathcal{S}} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\texttt{immediate}\\ \texttt{suspend}\ \mathcal{S}'\ \texttt{when}(\sigma)\end{array}, \mathcal{A}, \mathcal{C}\right\rangle}$$

$$(a_3)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\left\langle \mathcal{E}, C_{\mathcal{S}}, \begin{array}{l}\texttt{immediate}\\ \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\end{array}\right\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\texttt{immediate}\\ \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\end{array}, \{\}, \{C_{\mathcal{S}}\}\right\rangle}$$

$$(a_4)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad\wedge\quad \mathcal{C} \neq \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\left\langle \mathcal{E}, C_{\mathcal{S}}, \begin{array}{l}\texttt{immediate}\\ \texttt{suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\end{array}\right\rangle \overset{c}{\twoheadrightarrow} \left\langle \begin{array}{l}\texttt{immediate}\\ \texttt{suspend}\ \mathcal{S}'\ \texttt{when}(\sigma)\end{array}, \mathcal{A}, \mathcal{C}\right\rangle}$$

$$(a_5)\quad \frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad\wedge\quad \mathcal{C} = \{\} \quad\wedge\quad \langle \mathcal{E}, C_{\mathcal{S}}, \mathcal{S}\rangle \overset{c}{\twoheadrightarrow} \langle \mathcal{S}', \mathcal{A}, \mathcal{C}\rangle}{\langle \mathcal{E}, C_{\mathcal{S}}, \texttt{immediate suspend}\ \mathcal{S}\ \texttt{when}(\sigma)\rangle \overset{c}{\twoheadrightarrow} \langle \texttt{nothing}, \mathcal{A}, \mathcal{C}\rangle}$$

Figure 11 **Transition rules IV (strong suspension).**

Appendix 2: Compilation algorithm

The compilation algorithm for programs of the presented language extension to the intermediate format are given in Figure 12, Figure 13, and in Figure 14.

B Compilation Algorithm

```
function Compile($\mathcal{S}$)
begin
  ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileSurface(C0, st, $\mathcal{S}$)
  ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileDepth(C0, $\mathcal{S}$)
  return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
end
```

Figure 12 Pseudo code of function `Compile`.

```
function CompileSurface($c$, strt, $\mathcal{S}$)
begin
  switch $\mathcal{S}$
  case [nothing]:
    return ({}, {})
  case [$x$ = $\tau$]: # actions
    return ({strt $\Rightarrow$ next($x$) = $\tau$}, {})
  case [$\ell$: pause($C$)]: # pause
    return ({}, {strt $\Rightarrow$ next($\ell$) = true})
  case [if ($\gamma$) { $\mathcal{S}_1$ } else { $\mathcal{S}_2$ }]: # conditional
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileSurface($c$, strt $\wedge$ $\gamma$, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileSurface($c$, strt $\wedge \neg\gamma$, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [$\mathcal{S}_1$; $\mathcal{S}_2$]: # sequence
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileSurface($c$, strt, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileSurface($c$, strt $\wedge$ inst$_{\mathcal{S}_1}$, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [$\mathcal{S}_1$ || $\mathcal{S}_2$]: # parallel threads
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileSurface($c$, strt, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileSurface($c$, strt, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [suspend { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileSurface($c$, strt, $\mathcal{S}'$)
  case [$\ell$: immediate suspend { $\mathcal{S}'$ } when($\gamma$)]:
    ($\mathcal{A}^{\mathsf{data}}$, $\mathcal{A}^{\mathsf{ctrl}}$) := CompileSurface($c$, strt $\wedge \neg\gamma$, $\mathcal{S}'$)
    return ($\mathcal{A}^{\mathsf{data}}$, $\mathcal{A}^{\mathsf{ctrl}} \cup$ {strt $\wedge \gamma \Rightarrow$ next($\ell$) = true})
  case [abort { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileSurface($c$, strt, $\mathcal{S}'$)
  case [immediate abort { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileSurface($c$, strt $\wedge \neg\gamma$, $\mathcal{S}'$)
  case [clock ($C$) { $\mathcal{S}'$ }]: # clock declaration
    return CompileSurface($C$, strt, $\mathcal{S}'$)
end
```

Figure 13 Pseudo code of function `CompileDepth`.

```
function CompileDepth($c$, $\mathcal{S}$)
begin
  switch $\mathcal{S}$
  case [nothing]:
    return ({}, {})
  case [$x$ = $\tau$]: # actions
    return ({}, {})
  case [$\ell$: pause($C$)]: # pause
    return ({}, {$\ell \wedge C \wedge$ susp$_{\mathcal{S}}$($C$) $\Rightarrow$ next($\ell$) = true})
  case [if ($\gamma$) { $\mathcal{S}_1$ } else { $\mathcal{S}_2$ }]: # conditional
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [$\mathcal{S}_1$; $\mathcal{S}_2$]: # sequence
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileSurface($c$, term$_{\mathcal{S}_1}$, $\mathcal{S}_2$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [$\mathcal{S}_1$ || $\mathcal{S}_2$]: # parallel threads
    ($\mathcal{A}_1^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_1$)
    ($\mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_2^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}_2$)
    return ($\mathcal{A}_1^{\mathsf{data}} \cup \mathcal{A}_2^{\mathsf{data}}$, $\mathcal{A}_1^{\mathsf{ctrl}} \cup \mathcal{A}_2^{\mathsf{ctrl}}$)
  case [suspend { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileDepth($c$, $\mathcal{S}'$)
  case [$\ell$: immediate suspend { $\mathcal{S}'$ } when($\gamma$)]:
    ($\mathcal{A}^{\mathsf{data}}$, $\mathcal{A}^{\mathsf{ctrl}}$) := CompileDepth($c$, $\mathcal{S}'$)
    return ($\mathcal{A}^{\mathsf{data}}$, $\mathcal{A}^{\mathsf{ctrl}} \cup$ {strt $\wedge \gamma \Rightarrow$ next($\ell$) = true})
  case [abort { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileDepth($c$, $\mathcal{S}'$)
  case [immediate abort { $\mathcal{S}'$ } when($\gamma$)]:
    return CompileDepth($c$, $\mathcal{S}'$)
  case [clock ($C$) { $\mathcal{S}'$ }]: # clock declaration
    return CompileDepth($C$, $\mathcal{S}'$)
end
```

Figure 14 Pseudo code of function `CompileSurface`.

Competing interests

The authors declare that they have no competing interests.

Acknowledgments

We thank the German Research Foundation (DFG) for supporting this work.

Received: 27 February 2012 Accepted: 31 January 2013
Published: 10 April 2013

References

1. A Benveniste, P Caspi, S Edwards, N Halbwachs, P Le Guernic, R de Simone, The synchronous languages twelve years later. Proc. IEEE. **91**, 64–83 (2003)
2. G Berry, in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, ed. by G Plotkin, C Stirling, and Tofte M. The foundations of Esterel (MIT Press Cambridge, 1998), pp. 425–454
3. N Halbwachs, P Caspi, P Raymond, D Pilaud, The synchronous dataflow programming language LUSTRE. Proc. IEEE. **79**(9), 1305–1320 (1991)
4. K Schneider, The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009
5. G Berry, A hardware implementation of pure Esterel. Sadhana. **17**, 95–130 (1992)

6. F Rocheteau, N Halbwachs, in *Real-Time: Theory in Practice vol. 600 of LNCS*, ed. by J de Bakker, C Huizing, WP de Roever, and Rozenberg G. Implementing reactive programs on circuits: a hardware implementation of LUSTRE (Springer Mook, The Netherlands, 1992), pp. 195–208
7. G Berry, The constructive semantics of pure Esterel (1999). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.2076
8. K Schneider, in *Distributed and Parallel Embedded Systems (DIPES)*, ed. by Rammig F. A verified hardware synthesis for Esterel (Kluwer Schloß Ehringerfeld, 2000), pp. 205–214
9. K Schneider, in *Application of Concurrency to System Design (ACSD)*. Embedding imperative synchronous languages in interactive theorem provers (IEEE Computer Society Newcastle Upon Tyne, 2001), pp. 143–154
10. K Schneider, J Brandt, T Schuele, A verified compiler for synchronous programs with local declarations. Electron. Notes Theor. Comput. Sci. (ENTCS). **153**(4), 71–97 (2006)
11. YT Li, S Malik, Performance analysis of real-time embedded software. (Kluwer, The Netherlands, 1999)
12. G Logothetis, K Schneider, in *Design, Automation and Test in Europe (DATE)*. Exact high level WCET analysis of synchronous programs by symbolic state space exploration (IEEE Computer Society Munich, 2003), pp. 10196–10203
13. M Boldt, C Traulsen, R von Hanxleden, Worst case reaction time analysis of concurrent reactive programs. Electron. Notes Theor. Comput. Sci. (ENTCS). **203**(4), 65–79 (2008)
14. L Ju, B Khoa Huynh, A Roychoudhury, S Chakraborty, in *Design Automation Conference (DAC)*, ed. by S Sapatnekar. Timing analysis of Esterel programs on general purpose multiprocessors (ACM Anaheim, 2010), pp. 48–51
15. K Schneider, J Brandt, T Schuele, *Causality analysis of synchronous programs with delayed actions* (ACM, Washington, 2004), pp. 179–189
16. B Titzer, J Palsberg, in *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ed. by Y Paek, Gupta R. Nonintrusive precision instrumentation of microcontroller software (ACM Chicago, IL, 2005), pp. 59–68
17. J Brandt, K Schneider, in *Formal Methods and Models for Codesign (MEMOCODE)*, ed. by R Bloem, P Schaumont. Static data-flow analysis of synchronous programs (IEEE Computer Society Cambridge, 2009), pp. 161–170
18. L Carloni, K McMillan, A Sangiovanni-Vincentelli, Theory of latency-insensitive design. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (T-CAD). **20**(9), 1059–1076 (2001)
19. J Cortadella, M Kishinevsky, B Grundmann, in *Design Automation Conference (DAC)*, ed. by Sentovich E. Synthesis of synchronous elastic architectures (ACM San Francisco, 2006), pp. 657–662
20. S Krstic, J Cortadella, M Kishinevsky, J O'Leary, in *Formal Methods in Computer-Aided Design (FMCAD)*, ed. by A Gupta, Manolios P. Synchronous elastic networks (IEEE Computer Society San Jose, 2006), pp. 19–30
21. N Halbwachs, *Synchronous Programming of Reactive Systems*. (Kluwer, The Netherlands, 1993)
22. D Harel, A Naamad, The STATEMATE semantics of Statecharts. ACM Trans. Softw. Eng. Methodol. (TOSEM). **5**(4), 293–333 (1996)
23. S Malik, Analysis of cycle combinational circuits. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (T-CAD) **13**(7), 950–956 (1994)
24. N Halbwachs, F Maraninchi, in *Euromicro Conference*. On the symbolic analysis of combinational loops in circuits and synchronous programs (IEEE Computer Society Como, 1995)
25. J Brzozowski, CJ Seger, *Asynchronous Circuits*. (Springer, New York, 1995)
26. T Shiple, G Berry, H Touati, in *European Design Automation Conference (EDAC)*. Constructive analysis of cyclic circuits (IEEE Computer Society Paris, 1996), pp. 328–333
27. F Boussinot, SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France 1998
28. K Schneider, J Brandt, T Schuele, T Tuerk, in *Application of Concurrency to System Design (ACSD)*, ed. by J Desel, Watanabe Y. Maximal causality analysis, (IEEE Computer Society Saint-Malo, 2005), pp. 106–115
29. J Brandt, K Schneider, Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern , Germany 2011
30. O Tardieu, R de Simone, in *Formal Methods and Models for Codesign (MEMOCODE)*. Curing schizophrenia by program rewriting in Esterel (IEEE Computer Society San Diego, 2004), pp. 39–48
31. J Brandt, K Schneider, in *Software and Compilers for Embedded Systems (SCOPES) Volume 320 of* ACM International Conference Proceeding Series, ed. by Falk H. Separate compilation for synchronous programs (ACM Nice, 2009), pp. 1–10
32. K Chandy, J Misra, *Parallel Program Design*. (Addison-Wesley, Austin, 1989)
33. D Dill, in *Computer-Aided Verification (CAV), Volume 110 of* LNCS, ed. by R Alur, Henzinger T. The Murphi verification system (Springer New Brunswick, 1996), pp. 390–393
34. L Lamport, The temporal logic of actions. Technical Report 79. Digital Equipment Cooperation 1991
35. M Gemünde, J Brandt, K Schneider, in *Application of Concurrency to System Design (ACSD)*, ed. by L Gomes, V Khomenko, and Fernandes J. A formal semantics of clock refinement in imperative synchronous languages (IEEE Computer Society Braga, 2010), pp. 157–168
36. G Plotkin, A structural approach to operational semantics. Technical Report FN-19, DAIMI, Arhus, Denmark 1981
37. P Mosses, Formal semantics of programming languages. Electron. Notes Theor. Comput. Sci. (ENTCS). **148**, 41–73 (2006)
38. G Berry, L Cosserat, in *Seminar on Concurrency (CONCUR) Volume 197 of LNCS*, ed. by S Brookes, A Roscoe, and G Winskel. The Esterel synchronous programming language and its mathematical semantics (Springer Pittsburgh, 1985), pp. 389–448
39. S Tini, Structural operational semantics for synchronous languages. PhD thesis. University of Pisa, Italy, 2000
40. M Gemünde, J Brandt, K Schneider, in *Forum on Specification and Design Languages (FDL)*, ed. by K Morawiec, J Hinderscheit, and Ghenassia O. Schizophrenia and causality in the context of refined clocks (IEEE Computer Society Oldenburg, 2011), pp. 1–8
41. M Gemünde, J Brandt, K Schneider, in *High Level Design Validation and Test Workshop (HLDVT)*. Causality analysis of synchronous programs with refined clocks (IEEE Computer Society, 2011), pp. 25–32
42. M Gemünde, J Brandt, K Schneider, in *Formal Methods and Models for Codesign (MEMOCODE)*, ed. by L Carloni, Jobstmann B. Compilation of imperative synchronous programs with refined clocks (IEEE Computer Society Grenoble, 2010), pp. 209–218
43. G Berry, A quick guide to Esterel. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2222, 1997
44. G Berry, E Sentovich, ed. by T Margaria, Melham T. Correct Hardware Design and Verification Methods (CHARME), Volume 2144 of *LNCS* (Springer Livingston, 2001), pp. 110–125
45. B Rajan, R Shyamasundar, in *International Parallel and Distributed Processing Symposium (IPDPS),* Cancún. Multiclock ESTEREL: a reactive framework for asynchronous design (IEEE Computer Society Quintana Roo, 2000), pp. 201–209
46. B Rajan, R Shyamasundar, in *Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE/PSTV)*, ed. by T Bolognesi, Latella D. Modeling distributed embedded systems in multiclock Esterel (Kluwer Pisa, 2000), pp. 301–316
47. N Halbwachs, in *Formal Methods and Models for Codesign (MEMOCODE)*. A synchronous language at work: the story of Lustre (IEEE Computer Society Verona, 2005), pp. 3–11
48. T Gautier, P Le Guernic, L Besnard, in *Functional, Programming Languages and Computer Architecture, Volume 274 of* LNCS, ed. by G Kahn. SIGNAL, a declarative language for synchronous programming of real-time systems (Springer Portland, 1987), pp. 257–277
49. P Le Guernic, T Gauthier, M Le Borgne, C Le Maire, Programming real-time applications with SIGNAL. Proc. IEEE. **79**(9), 1321–1336 (1991)
50. P Le Guernic, JP Talpin, JC Le Lann, Polychrony for system design. J. Circuits Syst. Comput. (JCSC). **12**(3), 261–304 (2003)
51. D Potop-Butucaru, B Caillaud, A Benveniste, in *Application of Concurrency to System Design (ACSD)*. Concurrency in synchronous systems (IEEE Computer Society Hamilton, 2004), pp. 67–76
52. D Potop-Butucaru, B Caillaud, in *Application of Concurrency to System Design (ACSD)*. Correct-by-construction asynchronous implementation of modular synchronous specifications (IEEE Computer Society Saint-Malo, 2005), pp. 48–57

doi:10.1186/1687-3963-2013-3
Cite this article as: Gemünde *et al.*: **Clock refinement in imperative synchronous languages.** *EURASIP Journal on Embedded Systems* 2013 **2013**:3.