Schizophrenia and Causality in the Context of Refined Clocks

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

***Abstract*—Temporal refinement of synchronous programs is a desirable transformation in many design flows, in particular, to combine modules that work at different levels of abstraction. In synchronous languages, this refinement can be represented in the programs by means of sub-clocks. While this introduction leads to convenient temporal refinement techniques, it significantly complicates the classic causality and reincarnation problems that have to be handled in compilers for imperative synchronous languages. In this paper, we first generalize previous solutions to schizophrenia problems of determining and replicating affected code parts to multiple clocks. Second, our compilation algorithm extracts data dependencies between different clock domains and inserts explicit appropriate synchronization in the intermediate format. Finally, we illustrate the feasibility of our approach with the help of a running example and show the usage by a hardware translation.**

I. Introduction

The synchronous model of computation [1, 11] has shown to be a solid basis for the development of safety-critical embedded systems. It is based on a well defined and very natural notion of time, which is a key to the concise formal semantics of synchronous languages such as Esterel [3, 6], Lustre [12], Signal [8, 15] or Quartz [19]. This makes synchronous programs particularly attractive for formal verification [4, 21] and static analysis [16]. For this reason, efficient and provable correct synthesis [2, 17–19] to hardware circuits and *deterministic* software are available for developers.

The core of the synchronous model of computation is its *abstraction to discrete points of time*, which divides the execution of programs into *micro and macro steps* where variables change synchronously between macro steps and remain constant during micro steps. As a consequence, all threads of a program run in lockstep: they execute the micro steps of their macro step in the same variable environment, and automatically synchronize by definition at the end of the macro step. One often abstracts from this view by saying that micro steps are executed in zero time, and a macro step takes one unit of logical time.

While the synchronous model of computation has many advantages, it imposes unnecessary restrictions for programmers since there is no means to express the independence of threads in certain program locations. This phenomenon, where unnecessary synchronous lockstep or execution or dynamic workload of threads is still demanded, is often called *over-synchronization*. It usually occurs when the input signals of a system 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. We recently [10] introduced refined clocks, called sub-clocks, to add an explicit notion of independence that makes it possible for compilers to create desynchronized code without expensive analyses. They allow independent execution of parallel threads with the ability of synchronization. However, they preserve the advantages of the synchronous model of computation including determinism.

Dealing with different clocks is a common approach in system design. For synchronous languages, many concepts and methodologies have been proposed that deal with multiple clocks. A multi-clock extension to Esterel is proposed in [5], which models multi-clocked systems by single-clocked Esterel. In contrast to our extension, the clocks are provided by the environment. For determinism, the clocks must be generated deterministically, whereas our approach *generates* the clock ticks deterministically inside the system.

The synchronous data-flow languages Lustre and Signal also offer different clocks [8, 15]. In particular, the language Signal provides a mechanism called *oversampling*, which is quite similar to the considered clock refinement. However, since this paper addresses compilation problems that occur with imperative languages we refer to [10] for a more detailed discussion. Other references that are related to the considered problems are discussed in the appropriate sections.

This paper complements previous papers concerning clock refinement, which was originally proposed in [10]. A preliminary compilation scheme for such programs was presented in [9]. However, this scheme can only translate the control-flow of a program completely. In particular, schizophrenia and causality problems cannot be handled, so far.

This paper closes this gap and considers these classical problems in the context of refined clocks. Its main contribution is a compilation procedure that copes with both problems and translates a Quartz program to synchronous guarded actions with sub-clocks. To support the practicability of the translation, we show how hardware synthesis can benefit from these sub-clocks to generate a synchronous circuit.

The rest of this paper is structured as follows: In Section II, we explain clock refinement for imperative synchronous programs, the starting point of our translation. Section III focuses on the other end, synchronous guarded actions, which serve as the intermediate format in our compiler. Sections IV and V

present the core of the paper: they show how schizophrenia and causality problems are solved by our compilation procedure. Finally, Section VI shows how models in the intermediate format can be used for hardware synthesis before we conclude with a short summary in Section VII.

II. Synchrony and Clock Refinement

The synchronous model of computation [1, 11] divides the execution of a program into a sequence of macro steps [13]. In each of these steps, the system reads the inputs, performs some computation and finally produces the outputs. In theory, the semantics assume that the outputs appear at the very same instant when the inputs are read. In practice, the execution implicitly follows the data dependencies between the micro steps, and outputs have to be computed *fast enough* for the given application.

Imperative synchronous languages implement this 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 is therefore responsible for defining the end of a macro step. Thus, a macro step consists of all actions between two consecutive **pause** statements. Since all parallel threads are based on the same clock, they run in lock-step. All variables have a unique value per macro step and the assignments must therefore be executed according to their data dependencies.

As already stated in the introduction, over-synchronization is an undesired effect of the synchronous model of computation. *Clock refinement* can be used to avoid this problem. Its basic idea is illustrated with the help of two implementations of the *Euclidean Algorithm* to compute the greatest common divisor (GCD).

The first variant, which is given on the left-hand side of Figure 1, does not use refined clocks. The module reads its two inputs `a` and `b` in the first step and computes then iteratively the GCD with use of the local variables `x` and `y`. The computation steps are separated by the **pause** statement with label ℓ. Finally, the GCD is written to the output `gcd`. Apparently, a drawback of this implementation is that the computation is spread over several computation steps whose number depends on the input values, and each call to this module has to take care of the the data-dependent computation time.

The second variant uses refined clocks and is shown on the right-hand side of Figure 1. While the overall algorithm stays the same, the computation time is now hidden behind a declaration of a local clock. 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 on a **pause** statement of the outer clock and thus, the computation steps are not visible outside the module. As a consequence, each call to this module seems to be completed in a single step.

The refinement of clocks introduces arbitrary abstraction levels to the single-clocked synchronous model. Furthermore, the extension gives developers some flexibility for desynchronized implementations through unrelated clock refinements i. e. clocks which are declared in parallel threads. In general,

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

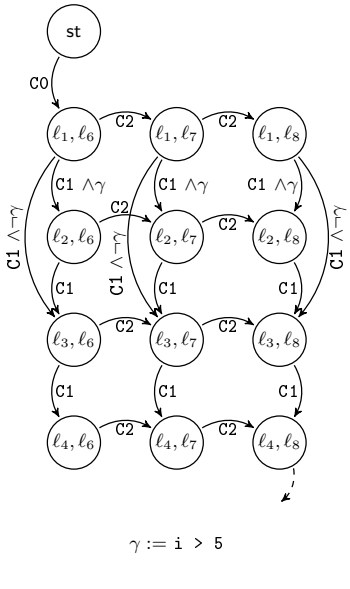
```
module GCD2
  (nat ?a, ?b, !gcd)
{
  clock(C1) {
    nat x = a, y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x − y;
      else
        next(y) = y − x;
      ℓ: pause(C1);
    }
    gcd = y;
  }
}
```

(b) Clock Refinement

Fig. 1. Greatest Common Divisor

```
module Example
   (nat ?i, o1, o2, o3)
{
  while(true) {
    nat x;
    clock(C1) {
      nat y;
      ℓ1: pause(C1);
      y = i;
      next (y) = y + 1;
      if(i > 5) {
        ℓ2: pause(C1);
        o1 = x + y;
      }
      ℓ3: pause(C1);
      ℓ4: pause;
      x = i;
      o2 = x;
      ℓ5: pause;
    }
  } || clock(C2) {
    ℓ6: pause(C2);
    o3 = o1;
    ℓ7: pause(C2);
    ℓ8: pause;
  }
}
```

(a) Parallel Clocks



(b) Finite-State Machine

Fig. 2. Running Example

the clock declarations form a *tree* and local clock declarations are not visible outside the module.

In the same way as micro steps are executed within a macro step, the substeps introduced by a refined clock are executed within a step of a higher (slower) clock. As shown in the example, variables can be declared for sub-clocks: they can change their value for every step of this clock. For unrelated clocks, the different substeps are not synchronized and they can be executed independently only with respect to data dependencies. Since variables at lower (faster) clock levels are not visible at higher levels, they are also not visible for unrelated clocks. Thus, communication between such clocks must take place through variables on higher levels.

As it can be seen in the example, the module's clock is not explicitly declared. We simply refer to this base clock by `C0` in the rest of the paper. In addition, 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 higher level (slower) than c_2. The relations $\succcurlyeq, \prec, \preccurlyeq$ are used accordingly. If two clocks c_1 and c_2 are independent, i. e. neither $c_1 \succcurlyeq c_2$ nor $c_1 \preccurlyeq c_2$ holds, we write $c_1 \# c_2$. This may hold if the two clocks are declared in different threads of a parallel statement.

The concepts and solutions which are discussed in this paper are illustrated by the running example shown in Figure 2 (a). The module takes the input `i` and produces the outputs `o1`, `o2`, and `o3` which are all natural numbers. The behavior of the module is described by two parallel threads, each thread defines a new sub-clock which divides steps of the module base clock `C0` to smaller execution steps. For these clocks, we have $\texttt{C0} \succ \texttt{C1}$, $\texttt{C0} \succ \texttt{C2}$ and $\texttt{C1} \ \# \ \texttt{C2}$. Additionally, the first thread is surrounded by a loop. The right part of the figure is explained later.

III. Intermediate Format

Our work is based on Averest[1], a framework for simulation, compilation, verification, and synthesis for the synchronous programming language Quartz [19]. Thereby, its compiler translates the source files to AIF (Averest Intermediate Format) files [7], which essentially consist of synchronous guarded actions. Thus, AIF abstracts from the complexity of the source language, because difficult interaction of preemption statements or reincarnations of local variables 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 compiler to various backend tools.

In analogy to the extension of the Quartz programs by sub-clocks, we also have to extend the intermediate format by sub-clocks. Instead of giving a full formalization, this section illustrates its structure and its semantics with the help of our running example in Figure 2 (a). Both clocks `C1` and `C2` are independent so that both threads only synchronize on **`pause`** statements of the common higher clock `C0`, which are in this example the **`pause`** statements with labels ℓ_4 and ℓ_8.

The behavior of the module can be represented by a finite-state machine (FSM). Just a small part of the complex FSM for the example is shown in Figure 2 (b). Its states are labeled with the labels of those **`pause`** statements that hold an active control flow in the state. The transitions are labeled with the conditions that must hold to change the state. The whole module initially starts in state `st`, where all **`pause`** statements are inactive. From there, the module switches to state (ℓ_1, ℓ_6) when a tick for clock `C0` occurs thus, both threads are started. From these labels, either the first or the second thread can do a substep, thus either `C1` or `C2` can tick and the resulting state depends on the labels that are reached by the step. Note that the next label of the first thread depends on the evaluation of the **`if`** condition. Finally, both threads synchronize at the **`pause`** statements with labels ℓ_4 and ℓ_8, which are both defined on the common clock `C0`. Thus, both threads can make independent substeps until they reach these labels. Then both go ahead synchronously by a tick of `C0`. The remaining FSM is omitted. In the view of the FSM, the clocks can be seen as inputs which are constrained by the overall semantics: a clock tick can only occur if all current labels are declared at least at this level. For example, the clock `C0` is not allowed to tick in state (ℓ_4, ℓ_6) because the second thread has to reach a label of clock `C0` before.

[1] `http://www.averest.org`

Figure 3 (a)-(d) show the guarded actions of the running example. In general, guarded actions either have the form $\gamma \Rightarrow x = \tau$ (immediate action) or $\gamma \Rightarrow \texttt{next}(x) = \tau$ (delayed action). γ is called the *guard* of the action, and the action is executed when it evaluates to `true`. An immediate action evaluates the right-hand side τ in the current step and sets the value immediately to x, while delayed actions transfer the value to the following step.

In addition to the transitions, our intermediate format also contains the clock constraints, which are shown in Figure 3 (e). They determine a clock *may* tick (note that a clock might not tick since we have independent threads). For the given example, the constraints basically reflects the labels which are attributed to a clock. The variation with $w_{\texttt{o1}}$ is explained later. In addition to the constraints which are explicitly given, the clocks must also preserve the clock tree, i. e. when `C0` ticks, also `C1` and `C2` must hold, because $\texttt{C0} \succ \texttt{C1}$ and $\texttt{C0} \succ \texttt{C2}$ holds. However, `C1` and `C2` can tick independently, because $\texttt{C1} \ \# \ \texttt{C2}$ holds.

IV. Schizophrenia

Schizophrenia problems are well-known for (imperative) synchronous languages with local variables. In the following, we first explain schizophrenia problems and their solutions. Second, we present a solution for them in the context of refined clocks.

A. Analysis of Schizophrenia

In a synchronous program, a macro step ranges over a finite number of statements in the source code. Therefore, it is possible that the scope of local variables is left and re-entered in the same macro step – a common situation in loops.

Figure 4 shows two examples in single-clocked Quartz. In both cases, the scope of the variable `x` is limited to the loop. In Example (a), the first execution step enters the loop and stops at the **`pause`** with label ℓ. The second step resumes from ℓ, the rest of the loop body is executed, due to the loop condition the loop is restarted, and the step ends again at the **`pause`** with label ℓ. Thus, the scope of `x` is left and entered in this step. Thereby, the assignment `x = true` affects the variable in the *old* scope, whereas the **`if`** statement reads the value of the *new* scope which is **`false`** because `x` is initialized when the scope is entered. Example (b) adds conditional **`pause`** statements[2], which allows the assignment `x = true` to be executed in the first or in the second part of the loop, depending on the input `i`. Thus, we have to distinguish the *old* and the *new* `x`. Nested loops can lead to an arbitrary finite number of *incarnations* of a variable.

Different solutions for this schizophrenia problems have been proposed in the past. The simplest solution, which was actually used in the first Esterel compilers, consists of duplicating loop bodies, i. e. replacing a loop **`while`**$(\sigma)\{S\}$ by the sequence **`while`**$(\sigma)\{S;$**`if`**$(\sigma)\{S\}\}$. However, this solution is expensive and it may be lead to an exponential

[2] This program is not causally correct in the sense of Esterel where this kind of schizophrenia is usually limited to parallel statements. However, the presented example is a valid example for Quartz.

$$
\begin{aligned}
\texttt{C0} \wedge \texttt{st} &\Rightarrow \texttt{next}(\ell_1) = \texttt{true}\\
\texttt{C1} \wedge \ell_1 &\Rightarrow \texttt{y} = \texttt{i}\\
(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell_1 &\Rightarrow \texttt{next}(\ell_2) = \texttt{true}\\
\neg(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell_1 &\Rightarrow \texttt{next}(\ell_3) = \texttt{true}\\
\texttt{C1} \wedge \ell_1 &\Rightarrow \texttt{next(y)} = \texttt{y} + 1\\
\texttt{C1} \wedge \ell_2 &\Rightarrow \texttt{next}(\ell_3) = \texttt{true}\\
\texttt{C1} \wedge \ell_2 &\Rightarrow \texttt{o1} = \texttt{x} + \texttt{y}\\
\texttt{C1} \wedge \ell_3 &\Rightarrow \texttt{next}(\ell_4) = \texttt{true}\\
\texttt{C0} \wedge \ell_4 &\Rightarrow \texttt{next}(\ell_5) = \texttt{true}\\
\texttt{C0} \wedge \ell_4 &\Rightarrow \texttt{x} = \texttt{i}\\
\texttt{C0} \wedge \ell_4 &\Rightarrow \texttt{o2} = \texttt{x}
\end{aligned}
$$

(a) Loop Body

$$
\begin{aligned}
\texttt{C1} \wedge \ell'_2 &\Rightarrow \texttt{next}(\ell'_3) = \texttt{true}\\
\texttt{C1} \wedge \ell'_3 &\Rightarrow \texttt{next}(\ell_4) = \texttt{true}\\
\texttt{C0} \wedge \ell_5 &\Rightarrow \texttt{next}(\ell'_1) = \texttt{true}\\
\texttt{C1} \wedge \ell'_1 &\Rightarrow \texttt{next(y)} = \texttt{y} + 1\\
(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell'_1 &\Rightarrow \texttt{next}(\ell'_2) = \texttt{true}\\
\neg(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell'_1 &\Rightarrow \texttt{next}(\ell'_3) = \texttt{true}\\
\texttt{C1} \wedge \ell'_1 &\Rightarrow \texttt{y} = \texttt{i}\\
\texttt{C1} \wedge \ell'_2 &\Rightarrow \texttt{o1} = \texttt{x}' + \texttt{y}
\end{aligned}
$$

(b) Renamed Loop Body

$$
\begin{aligned}
\texttt{C0} \wedge \texttt{st} &\Rightarrow \texttt{next}(\ell_6) = \texttt{true}\\
\texttt{C2} \wedge \ell_6 &\Rightarrow \texttt{next}(\ell_7) = \texttt{true}\\
\texttt{C2} \wedge \ell_7 &\Rightarrow \texttt{next}(\ell_8) = \texttt{true}\\
\texttt{C2} \wedge \ell_6 &\Rightarrow \texttt{o3} = \texttt{o1}
\end{aligned}
$$

(c) Other Actions

$$
\begin{aligned}
\texttt{C1} \wedge \ell_2 &\Rightarrow w_{\texttt{o1}} = \texttt{true}\\
\texttt{C1} \wedge \ell'_2 &\Rightarrow w_{\texttt{o1}} = \texttt{true}\\
\neg(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell_1 &\Rightarrow w_{\texttt{o1}} = \texttt{true}\\
\neg(\texttt{i} > 5) \wedge \texttt{C1} \wedge \ell'_1 &\Rightarrow w_{\texttt{o1}} = \texttt{true}\\
\texttt{C0} \wedge \ell_4 &\Rightarrow w_{\texttt{o1}} = \texttt{true}
\end{aligned}
$$

(d) Causality Actions

$$
\begin{aligned}
\texttt{C0} &\rightarrow \texttt{st} \vee \ell_4 \vee \ell_5 \vee \ell_8\\
\texttt{C1} &\rightarrow \ell_1 \vee \ell_2 \vee \ell_3 \vee \ell'_1 \vee \ell'_2 \vee \ell'_3\\
\texttt{C2} &\rightarrow \ell_6 \wedge w_{\texttt{o1}} \vee \ell_7
\end{aligned}
$$

(e) Clock Constraints

```
nat x';
clock(C1) {
  nat y;
  ℓ'1: pause(C1);
  y = i;
  next(y) = y + 1;
  if(i > 5) {
    ℓ'2: pause(C1);
    o1 = x' + y;
  }
  ℓ'3: pause(C1);
  ℓ4: pause;
  x' = i;
  o2 = x';
  ℓ5: pause;
}
```

(f) Renamed Loop Body

Fig. 3. Guarded Actions of Running Example

```
module A (nat !o)
{
  while (true) {
    bool x;
    if (x)
      o = 1;
    ℓ: pause;
    x = true;
  }
}
```

(a) Reentering a Scope

```
module B (bool ?i, nat !o)
{
  while (true) {
    bool x;
    if (x)
      o = 1;
    if (i)
      ℓ1: pause;
    x = true;
    if (!i)
      ℓ2: pause;
  }
}
```

(b) Schizophrenic Statements

Fig. 4. Schizophrenic Quartz Programs

blow-up in the worst case for nested loops. A better source-code transformation for Esterel was proposed in [22], where also the local variables and other statements are duplicated, but not the whole loop. Similarly, the Quartz compiler [7] duplicates the variables and assignments of the first macro step. Finally, another approach was proposed by Yun et al. [23], which tackles the problem on basis of graph reachability, but their solution is not applicable when hardware synthesis is desired. Further solutions are discussed in [21].

The solutions known for Esterel have to be extended to handle delayed assignments in Quartz: In Esterel, the value of a variable from the last step can be accessed, whereas Quartz utilizes delayed assignments to set a variable for the next step. E.g. the duplication of the loop body is not sufficient, because a delayed assignment to a local variable can be executed when the loop body is left, but with the duplication, the same scope can be entered one step after, i. e. when the delayed assignment assigns the value. Without modifications, it will affect the variable in the new scope. To apply the duplication technique, a second copy of the loop body would be needed, which is even more expensive. The Quartz compiler solves this by disabling the delayed assignments to local variables at the end of a loop.

B. Handling Schizophrenia for Refined Clocks

Apparently, schizophrenia problems also occur in the context of refined clocks, and they may be even more complex: the first step of the variable's scope may be divided into substeps of a lower clock. In this case, not only the assignments of the first step, but also its internal control-flow must be duplicated. In Figure 2 (a), the scope of the variable `x` begins with a loop. Thus, a step of clock `C0` starting from the label ℓ_5 re-enters the loop and eventually ends at the label ℓ_4. The whole step is divided by substeps of clock `C1`. Therefore, all substeps after re-entering the loop must consider `x` of the *new* scope.

As already mentioned, schizophrenia is handled by the Quartz compiler by duplicating affected variables and assignments during compilation. Delayed assignments at the end of the loop are disabled. However, disabling the delayed assignments is not an easy task when refined clocks come into play: when a delayed assignment is executed, it is not known if the loop is restarted in the same step because substeps can hide this information. This is in contrast to the single-clock case, where this can be expressed by a condition. Another issue arises with delayed assignments to variables of lower clocks in the first step of a loop. If the whole first step is duplicated, such assignments may be executed to the original variable or to the copy. However, this information may be again hidden by substeps. We define restrictions for our language extension to tackle this problems: the restrictions (1) are fulfilled by the examples like the GCD and (2) still allows us to compile the code with reasonable effort. The restrictions are:

1) A **`pause`** statement of the loop's clock must occur at the end of the loop.
2) Each clock block must start with a **`pause`** statement of the declared clock before the first assignment.

The first restriction simply forbids delayed assignments at the end of the loop, i. e. such assignments which are disabled by the single-clock Quartz compiler. However, it does not remove the need to duplicate the first step of the loop, because delayed assignments can still occur one step before. However,

the second restriction makes the duplication of the first step easier, because schizophrenia problems do no longer occur for the local variables of the lower clocks. Therefore, there is no need to duplicate the variables and delayed assignment refer only to the original variable. We think that these restrictions do not really impose restrictions to the developer, but make the compilation manageable. The second restriction is just a variation, because substeps are not visible outside the module. We describe the compilation of a loop based on this restrictions in the following.

The basic idea is illustrated in Figure 5. The leftmost part shows the loop as it occurs in the source code whereas the rightmost part shows the structure of the compiled loop. The behavior of the loop body is compiled as it is, but when the loop is restarted, we have to care about local variables whose scopes are re-entered. Therefore, the *first* step at the beginning of the loop is determined and all occurrences of the local variables with the loop's clock are renamed in this step to a new unique name. The renamed step is used for a restart instead of the real first step of the loop. The renaming ensures that local variables whose scopes are entered by a loop are not confused with the variables having the same name whose scope is left. At the end of the step that re-enters the loop, the normal behavior of the loop body must be adopted. This is solved by *docking* the first step to the remaining loop body.

As the running example in Figure 2 (a) shows, the first step of a loop can be divided into steps of a lower clock. The classic compile algorithm re-enters the loop and stops at the first **pause** statement that is reached. However, we need to respect the clock of the **pause** statements and stop at the first **pause** statement with the loop's clock.

The existing compile algorithm for Quartz is based on some control-flow predicates [19, 21] which are crucial for the correctness. Especially, the predicate inst_S is of interest for the compilation of loops. It expresses for each statement S under which condition the statement is instantaneous, i. e. it is executed within an execution step. the condition holds for the whole step. However, it is not possible to express this condition for refined clocks, because a step for clock c may be divided by smaller steps and thus the condition may depend on variables of this lower clock which can change their value during the step of c. Therefore, the existing compile scheme can not be directly used for handling schizophrenia.

The presented solution for compiling loops in Quartz programs with refined clocks combines the original algorithm [7] with a graph reachability approach that is similar to the one of [23]. However, this approach just detects schizophrenia problems and does not address refined clocks. We want to detect and duplicate statements that are affected by schizophrenia. The algorithm uses the other predicates strt_S, which holds when a statement is started, and term_S, which holds when a statement terminates. To compile the loop statement $S \equiv$ **while**$(\sigma)\{S'\}$ with the clock c_S, i. e. all clocks inside S' are lower, we perform the following steps:

1) Compile body S' with start condition $\mathsf{strt}_{S'} = \mathsf{strt}_S \wedge \sigma$. The result is a set of guarded actions $\mathcal{A}_S$ and a termination condition $\mathsf{term}_{S'}$.

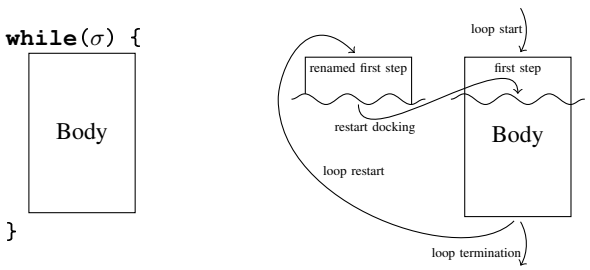


Fig. 5. Compilation of Loops

2) Create new body S'_R by renaming all occurrences of local variables of clock c_S in body S' and also all labels of **pause** statements related to clocks lower than c_S. The renaming is not performed for the target variables of delayed assignments.
3) Compile renamed body S'_R with the start condition $\mathsf{strt}_{S'_R} = \mathsf{term}_{S'} \wedge \sigma$ to a set of guarded actions $\mathcal{A}_{S'_R}$. The condition $\mathsf{strt}_{S'_R}$ expresses a restart of the loop.
4) Build the dependency graph of the control-flow of $\mathcal{A}_{S'_R}$. Therefore, consider all guarded actions that assign labels. Determine all labels $\mathcal{L}$ that can be reached from the start condition $\mathsf{strt}_{S'_R}$ by only passing labels with clocks lower than c_S. These are the labels of the substeps of the first step of the loop body.
5) Take all guarded actions $\mathcal{A}_{RS}$ from $\mathcal{A}_{S'_R}$ that have a label of $\mathcal{L}$ in its guard.
6) Return the guarded actions $\mathcal{A}_S \cup \mathcal{A}_{RS}$ and the condition $\mathsf{term}_S = \mathsf{term}_{S'} \wedge \neg\sigma$ for the whole loop.

Note, that only labels of clocks declared inside the loop are renamed and labels with a clock that is higher or equal to c_S are kept. This ensures that the first step of the re-entrance ends at a label of the original body and all further steps are performed on the original body.

Consider our running example in Figure 2 (a), and in particular, the body of its loop. First, we compile the body and get the guarded actions shown in Figure 3 (a). For the re-entering of the loop, we perform the previously described renaming, which results in the body shown in Figure 3 (f), where e. g. the variable `x` is renamed to `x'`, and the label ℓ_1 is renamed to ℓ'_1. Note that the label ℓ_4 is not renamed because it does not belong to the local clock `C1`. The compilation of the renamed body results in the guarded actions shown in Figure 3 (b). The actions which cannot be reached in the first step are already removed by the graph reachability analysis.

V. Causality

The second problem for compiling synchronous languages are cyclic causal dependencies. In this section, we first examine causality problems of traditional synchronous languages. Then, we explain a new kind of causality introduced by refined clocks and how that is resolved by the compiler.

A. Causality in Single-Clocked Synchronous Languages

In synchronous languages, all variables have a unique value in each macro step, and all executions must be consistent with

```
module Caus1 (nat ?a, x, y)
{
  ℓ1: pause;
  y = a + x;
  x = 2 * a;
  ℓ2: pause;
}
```

(a) Causality 1

```
module Caus2 (nat ?a, x, y)
{
  ℓ1: pause;
  if (y > 2)
    x = 2 * a;
  y = a + 1;
  ℓ2: pause;
}
```

(b) Causality 2

Fig. 6. Causality in the Synchronous Model

these values. Consider the module in Figure 6 (a), where in the step between the two **pause** statements the variables x and y are assigned, whereas y is assigned *before* x in the source code but the value of y depends on the value of x. However, the variable x has exactly one value for a step, and the value that is assigned to x must be the same which is used to calculate the value of y. It can be read as a set of equations which must be fulfilled for the step. A second module, which is given in Figure 6 (b), also assigns x and y, and the execution of the assignment to x depends on the value of y. Again, there is exactly one value for each variable and y must be computed before it can be decided whether x is assigned or not.

The assignments must be executed according to their dependencies to follow the semantics. Some synthesis backends, e. g. for sequential software, statically orders the actions appropriately, while e. g. synchronous hardware circuits can handle causality dynamically. This causality is restricted to an execution step and does not influence following steps.

B. Causality for Refined Clocks

As one might expect, clock refinement adds more complexity to causality. Variables still have one value for one reaction step, but, the *duration* of a step depends now on the clock of the variable. The new kind of causality that is introduced by refined clocks is shown in our running example in Figure 2 (a). The variable o1 is declared on clock C0 and it is written by the first and read by the second thread. There is exactly one value for a step of C0. However, the first thread divides this step into smaller steps of clock C1, and the second thread divides the step into (unrelated) smaller steps of clock C2. According to the synchronous model of computation, there is a read-after-write (RAW) dependency, i. e. the value written to x by the first thread must be the same as the value read by the second thread, but both actions may occur in different substeps.

This dependency must be preserved by all synthesis backends for all target platforms. If the clocks of the modules are mapped to different *clock* signals in circuits, the causality cannot be resolved at run-time anymore (see also Section VI), because dependencies now involve several (sub-)steps so that signal propagation in a synchronous circuit cannot handle it automatically. Therefore, our compilation algorithm handles causality problems due to clock refinement statically by explicitly modeling the RAW dependencies similar to [20]. For instance in the example, we have to delay the read of x in the second step of our example until the first thread has executed all substeps up to an assignment to x or it is known that x is not written in this C0 step.

Since, the problem can only occur for parallel threads, we concentrate on the compilation of the statement $S_1 \,||\, S_2$ to generate guarded actions:

1) Compile S_1 and S_2 to guarded actions $\mathcal{A}_{S_1}$ and $\mathcal{A}_{S_2}$.
2) Check for variables that are written in one thread and read by the other and that are read/written by actions which contain unrelated clocks in their guards.
3) For such a variable x, introduce a new variable $w_{\mathtt{x}}$ that indicates if x already got its value for the current step.
4) Add guarded actions for $w_{\mathtt{x}}$ with the same guards as the actions that writes x.
5) Add guarded actions for $w_{\mathtt{x}}$ with the labels so that no action that writes x is reachable in the common step.
6) Add $w_{\mathtt{x}}$ to the clock constraints for all labels that possibly read x.

The guarded actions for the running example which are generated to solve the causality problem are shown in Figure 3 (d). The variable $w_{\mathtt{o1}}$ enforces an explicit synchronization of both threads due to the RAW dependency on o1. Guarded actions to set $w_{\mathtt{o1}}$ whenever o1 is written are added. Additionally, an action is added to set $w_{\mathtt{o1}}$ when it is ensured that no action will write o1. This is the case in the example when the **if** clause is not taken, thus, the action has the guard $\neg(i > 5) \wedge \mathtt{C1} \wedge \ell_1$. The clock constraints in Figure 3 (e) are adjusted to wait on the **pause** with label ℓ_6 until $w_{\mathtt{o1}}$ is set.

Generally, the causality could be also implicitly encoded in the semantics of the guarded actions. However, the problem has to be solved eventually in the design flow so that we decided to introduce this explicit dependency by the compiler. Hence, the problem has not to be considered by further processing. In particular, hardware synthesis is straightforward, which is shown by the following section.

VI. Hardware Synthesis

We explained in the previous sections how schizophrenia and causality problems can be resolved for synchronous programs with refined clocks during a translation to synchronous guarded actions. This section explains how the resulting guarded actions can be translated to a hardware description. Thereby, we generate for a module of the intermediate format a corresponding Verilog description. The translation also shows that resolving the problems discussed before makes the synthesis from this intermediate format relatively simple.

The interface of our synthesized hardware module complies with the interface of the original Quartz module enhanced by three additional signals, the inputs clock and reset and an output C0. Thereby, clock is the hardware clock driven from the environment to trigger reactions of the module (thus, its *clock* in the usual sense). C0 is the module clock of the synchronous system, and it is determined by the module itself. It indicates whenever the outputs are available and new inputs can be read by the module. The signal reset is needed to initialize the circuit at the beginning.

In the hardware translation we have to distinguish between labels and other variables of the data-flow. First, consider the labels and assume that for a label ℓ the following guarded actions exist (which can be only delayed actions):

$$\begin{aligned}\gamma_1 &\Rightarrow \texttt{next}(\ell) = \tau_1\\ \gamma_2 &\Rightarrow \texttt{next}(\ell) = \tau_2\\ &\dots\\ \gamma_n &\Rightarrow \texttt{next}(\ell) = \tau_n\end{aligned}$$

The guarded actions are combined to a single expression that determines the value of the label for the next execution step. Then, this expression can be directly translated to hardware as the value for a register.

$$\texttt{next}(\ell) = \begin{cases} \tau_1 & : \quad \gamma_1\\ \tau_2 & : \quad \gamma_2\\ \vdots & \quad \vdots\\ \tau_n & : \quad \gamma_n\\ \ell \wedge \neg \mathsf{clk}\ (\ell) & : \quad \mathsf{default}\end{cases}$$

The expression can be expressed in Verilog by the ternary `:?`-operator. The default case of the expression is not straightforward since it cannot be directly derived from the guarded actions. If no delayed guarded action assigns a value for the next step, the label gets the value $\ell \wedge \neg \mathsf{clk}\ (\ell)$, which means that the value of ℓ is kept as long as the clock of ℓ does not tick. This can happen in parallel threads with unrelated clocks as it is also shown in the running example: when only clock `C1` ticks, the labels of clock `C2` do not change and keep their values until `C2` ticks.

The assignments to other variables of the data-flow can be translated in a similar way. However, we have to take care of immediate and delayed assignments: the value of the current execution step according the clock of a variable and a delayed value must be stored. This results in two registers for each variable in the worst case. (Again, this can be easily optimized in many cases.) Thus, assume that we have the following guarded actions for a variable `x`:

$$\begin{aligned}\gamma_1^i &\Rightarrow \quad \texttt{x} = \tau_1^i & \qquad \gamma_1^d &\Rightarrow \texttt{next}(\texttt{x}) = \tau_1^d\\ \gamma_2^i &\Rightarrow \quad \texttt{x} = \tau_2^i & \qquad \gamma_2^d &\Rightarrow \texttt{next}(\texttt{x}) = \tau_2^d\\ &\dots & &\dots\\ \gamma_n^i &\Rightarrow \quad \texttt{x} = \tau_n^i & \qquad \gamma_m^d &\Rightarrow \texttt{next}(\texttt{x}) = \tau_m^d\end{aligned}$$

The immediate and the guarded actions are combined separately, each group in the same way as the labels. The value for the default case is derived from the declaration of the variable: event variables are reset and memorized variables retain their previous values.

Finally, we need to translate the clock constraints, which is the most interesting part of the synthesis. The signals `C0_RDY`, etc. are the constraints for the clocks as they are given by the compiler and describe when a clock may tick. The definitions of the signals `C0`, etc. describe the relation of the clocks given by the clock tree and they enable a clock as soon as possible (ASAP). As we have resolved schizophrenia and causality problems in the course of our translation to the intermediate format, we can now enable the clocks whenever this is possible. Note, that this is not the case when the causality is not explicitly resolved because one clock may have to wait for a value of a variable. The final Verilog code for our running example can be seen in Figure 7.

VII. Conclusions

This paper discusses how the typical problems of compiling (imperative) synchronous languages, namely schizophrenia and causality, can be solved in the context of refined clocks. A solution for dealing with schizophrenia is presented by duplicating the step when a loop is re-entered. This approach is based on some restrictions which are defined in the paper: they keep refined clocks usable and make the compilation manageable. The solution is similar to handling schizophrenia in our compiler for single-clocked Quartz. However, more effort must be spent to duplicate also control-flow related to local clock declarations. The new kind of causality which is introduced especially by unrelated refinements of a clock is tackled by explicitly adding the imposed data dependencies. Both problems are illustrated by a running example. Finally, we showed the usefulness of the presented solutions by applying a straightforward hardware synthesis to the running example.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] G. Berry. A hardware implementation of pure Esterel. *Sadhana*, 17(1):95–130, March 1992.

[3] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[4] G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/, July 1999.

[5] G. Berry and E. Sentovich. Multiclock Esterel. In T. Margaria and T.F. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 110–125, Livingston, UK, 2001. Springer.

[6] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[7] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[8] T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL, a declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 257–277, Portland, Oregon, USA, 1987. Springer.

[9] M. Gemünde, J. Brandt, and K. Schneider. Compilation of imperative synchronous programs with refined clocks. In L. Carloni and B. Jobstmann, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 209–218, Grenoble, France, 2010. IEEE Computer Society.

[10] M. Gemünde, J. Brandt, and K. Schneider. A formal semantics of clock refinement in imperative synchronous languages. In L. Gomes, V. Khomenko, and J.M. Fernandes, editors, *Conference on Application of Concurrency to System Design (ACSD)*, pages 157–168, Braga, Portugal, 2010. IEEE Computer Society.

```
module Example (
       input clock,
       input reset,
       output C0,
       input [4:0] i,
       output [4:0] o1, o2, o3
   );

   wire C1, C2, C0_RDY, C1_RDY, C2_RDY;
   reg st, l1, l1_1, l2, l2_1, l3,
      l3_1, l4, l5, l6, l7, l8, pre_wo1;
   wire[4:0] x, y, wo1;
   reg [4:0] pre_x, pre_y, pre_o1,
      pre_o2, pre_o3, next_y;

   assign C0_RDY = (st || l4 || l5 || l8);
   assign C1_RDY = l1 || l2 || l3 || l1_1 ||
      l2_1 || l3_1;
   assign C2_RDY = l6 && wo1 || l7;

   assign C0 = C0_RDY && ! C1_RDY && ! C2_RDY;
   assign C1 = C1_RDY || C0;
   assign C2 = C2_RDY || C0;

   assign x = (C0 && l4) ? i : pre_x;
   assign y = (C0 && st) ? i : (C1 ? next_y : pre_y);
   assign x_1 = 0;

   assign o1 = (C1 && l2) ? (x + y) : ((C1 && l2_1) ?
      x_1 + y : pre_o1);
   assign o2 = (C1 && l4) ? x : pre_o2;
   assign o3 = (C2 && l6) ? o1 : pre_o3;

   assign wo1 = (C1 && l2) ? 1 : ((C1 && l2_1) ? 1 :
      ((!(i > 5) && C1 && l1) ? 1 :
      ((!(i > 5) && C1 && l1_1) ? 1 :
      ((C0 && l4) ? 1 :
      pre_wo1 && !C0))));

   always @(posedge clock) begin
      if (reset) begin
         st <= 1; l1 <= 0; l2 <= 0;
         l3 <= 0; l4 <= 0; l5 <= 0;
         l6 <= 0; l7 <= 0; l8 <= 0;
         l1_1 <= 0; l2_1 <= 0; l3_1 <= 0;
         next_y <= 0;
         pre_y <= 0; pre_x <= 0;
         pre_o1 <= 0; pre_o2 <= 0;
         pre_o3 <= 0; pre_wo1 <= 0;
      end /* if */
      else begin
         st <= st && !C0;

         l1 <= (C0 && st) ? 1 : (l1 && !C1);
         l2 <= ((i > 5) && C1 && l1) ? 1 : (l2 && !C1);
         l3 <= (!(i > 5) && C1 && l1) ? 1
            : ((C1 && l2) ? 1 : (l3 && !C1));
         l4 <= (C1 && l3) ? 1
            : ((C1 && l3_1) ? 1 : (l4 && !C0));
         l5 <= (C0 && l4) ? 1 : (l5 && !C1);
         l6 <= (C0 && st) ? 1 : (l6 && !C2);
         l7 <= (C2 && l6) ? 1 : (l7 && !C2);
         l8 <= (C2 && l7) ? 1 : (l8 && !C0);
         l1_1 <= (C0 && l5) ? 1 : (l1_1 && !C1);
         l2_1 <= ((i > 5) && C1 && l1_1) ? 1 :
            (l2_1 && !C1);
         l3_1 <= (!(i > 5) && C1 && l1_1) ? 1 :
            ((C1 && l2_1) ? 1 : (l3_1 && !C1));

         next_y <= (C1 && l1) ? y + 1 : C1 ?
            ((C1 && l3_1) ? y : y) : next_y;

         pre_y <= y;
         pre_x <= (C1 && l3_1) ? x_1 : x;
         pre_o1 <= o1;
         pre_o2 <= o2;
         pre_o3 <= o3;
         pre_wo1 <= wo1;
      end /* else */
   end /* always */
endmodule /* Example */
```

Fig. 7. Verilog Synthesis

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[14] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[15] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):261–304, June 2003.

[16] Y.-T.S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.

[17] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[18] F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, Bonas, France, 1991.

[19] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[20] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.

[21] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.

[22] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.

[23] J.-H. Yun, C.-J. Kim, S. Seo, T. Han, and K.-M. Choe. Refining schizophrenia via graph reachability in Esterel. In R. Bloem and P. Schaumont, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 18–27, Cambridge, Massachusetts, USA, 2009. IEEE Computer Society.