A Formal Semantics of Clock Refinement in Imperative Synchronous Languages

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

Abstract—The synchronous model of computation divides the execution of a program into an infinite sequence of so-called macro steps, which are further divided into finitely many micro steps. Since all threads of a program are forced to run in lockstep, programmers have no means to express the independence of parallel threads, which leads to a phenomenon called over-synchronization. In this paper, we therefore propose a generalization of the synchronous model of computation by means of refined clocks, which divide a macro step into finer grained steps that themselves consist of micro steps. In particular, we present a structural operational semantics of subclocks and prove that the internal asynchrony given by subclocks still preserves input/output determinism.

I. INTRODUCTION

Synchronous languages [1] like Esterel [3, 7], Lustre [14] or Quartz [26] 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 be directly executed on simple micro-controllers without having the need to use complex operating systems. In addition, synchronous programs can be straightforwardly translated to hardware circuits [2, 23, 26], which recommends synchronous languages for the use in hardware-software co-design. Furthermore, the concise formal semantics of synchronous languages makes them particularly attractive for formal verification of the correctness of the programs as well as of the used compilers [4, 24, 25, 27]. 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 [18].

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 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 must be executed in a causal ordering so that there are no read-after-write conflicts. Compilers check whether such a causal ordering exists, and reject programs as non-constructive programs if a dynamic scheduling of micro steps along their data dependences is not possible. 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. One often abstracts from this view by saying that micro steps are executed in zero time, and a macro step takes one amount of logical time.

Obviously, the synchronous model of computation is very strict and enforces deterministic concurrency, which has many advantages in system design, e.g. to avoid Heisenbugs [30] and to allow compile-time analyses e.g. on WCET. At the same time, however, it imposes tight restrictions on compilers and programmers, since there is no means to express the independence of threads in certain program locations. This phenomenon where synchronous lockstep execution of threads is still demanded even though this would not be necessary to ensure program correctness is often called 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 [8], adding an explicit notion of independence makes it possible for compilers to create desynchronized code without expensive analyses.

There are many reasons why a programmer may want to split a macro step of an already written program into further (macro) steps. However, even such local modifications of a component are most often problematic since they lead to a different temporal behavior that potentially modifies the global behavior of the entire system. For this reason, new design methods like latency-insensitive [9] designs or elastic synchronous systems [10, 15] have been developed to maintain the synchronous computation between modules in case the timing of one of the modules has been changed. An alternative is the refinement of the base clock of such a module while retaining its external input/output behavior. Using refinements of the base clock, it is possible to replace a code segment by another one having a different
temporal behavior. For instance, it is possible to replace an instantaneous/combinational multiplication by a sequential one. Also, it becomes possible to exchange components with functionally equivalent ones running at higher clock speeds. Obviously, refinements make component-based design much more convenient.

Another desirable feature that requires further temporal abstraction layers is given by function calls that must be executed within a micro step: For example, assume that the greatest common divisor of two integers is required in a program expression. As such non-primitive recursive functions require data-dependent loops, this is not possible since macro steps must be unrolled at compile time into finitely many micro steps. 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 be explicitly 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 generalization of the single-clocked synchronous model of computation by refined clocks, which can be used to solve all the above mentioned problems, was recently proposed in a seminal paper [12]. The main contribution of this paper is a structural operational semantics of our language extension and a theorem that states that refined clocks preserve the input/output determinism\(^1\) of the single-clocked synchronous model, while providing additional degrees of freedom for the internal implementation and execution.

The rest of this paper is structured as follows: Section II first presents our basic idea of clock refinement, before we sketch our language extension. Section III relates our approach to previous work. Section IV gives a formal semantics for our extension, which is subsequently used to prove an important property of our extension, namely that the input/output determinism is preserved. Section V explains the application of the formal semantics by an example. Finally, we conclude with a short summary in Section VI.

**II. CLOCK REFINEMENT**

In a seminal paper [12], we recently proposed subclocks as an extension to the imperative single-clocked synchronous

\(^1\) Input/output determinism ensures that the same input traces lead to the same output traces.

language Quartz [26], which is a descendnet of Esterel [3]. The basic idea is to allow the developer to divide a macro step into a sequence of finer grained macro steps, which may consist themselves of micro steps or further refined macro steps. However, we do not make this abstraction visible to the interface/environment of modules. The module interface has the same timing behavior, while its internal implementation has more freedom due to its internal subclocks. One instant on a clock is divided into some smaller steps of the lower clock as shown in Figure 1 (b) in contrast to the single-clocked counterpart shown in Figure 1 (a). As also shown, the communication of the system with the environment remains only possible at the highest clock, where the variables keep its value for the whole step. However, variables of subclocks can have multiple values during a step on a higher clock, but the variables are not visible to the higher clock. Similar to the distinction of micro and macro steps, the computation of one step is hidden to the higher clock, and only the result is visible. The advantage compared to micro and macro steps is that the clock hierarchy provided hereby can be arbitrarily deeply nested. Note that the number of substeps is dynamic; it only needs to be finite to ensure reactivity, which is generally a desirable property.

It is also possible to refine a clock by multiple unrelated subclocks. This leads to a tree of clocks shown in Figure 2 (a). Thereby, \(C_0\) is the clock exposed at the interface of the synchronous module. This clock is internally refined by other ones, as it is shown in the tree. Only the marked branch of the tree provides the clocks shown in the trace in Figure 1 (b). Unrelated clock domains must not share variables, but they can communicate over variables declared on common superclocks. Thereby, they can run independently until a clock tick of the common superclock arises, which also enforces a synchronization of all its subclocks. The whole model gives the implementation some internal flexibility for desynchronized implementations through unrelated subclocks while the synchronous interface to the outer environment can be kept at the same time. Since subclocks are not visible outside of the module, ticks cannot be provided by the environment for them, which emphasizes the view of a logical refinement instead of a multi-clock extension.

In the following, we only give a short overview of the language extension, which is required for the understanding of the rest of this paper. More explanations to highlight design decisions and consequences can be found in [12].

To make subclocks accessible to developers, we extend Quartz [26] by a new statement \(\text{clock}(C) \{S\}\), which declares a subclock \(C\) and determines the scope \(S\) where this subclock is visible and can be referenced. The only statement that directly addresses the new clock is a labeled pause statement \(\text{pause}(C)\), which enforces a synchroniza-
tion with respect to this clock $C$. Thereby, parallel running threads can synchronize at this clock as with any other clock.

The clock tree can be directly derived from the source code and it ensures that unrelated clocks are never visible in the same scope. Allowing arbitrary clock declarations like variables would destroy this assumption, and parallel threads could request a synchronization for unrelated clocks, which leads to potential deadlocks. Furthermore, by construction, the scope of a subclocked local variable is always smaller than the scope of its clock. This ensures that the variables bound to a subclock are not visible outside of their subclock domain.

Immediate assignments to variables change the value of the corresponding variable in the current step, while delayed assignments commit the result in the following step, where ‘following’ is defined by the clock of the target variable. Together with the visibility of variable declarations, it is guaranteed that each expression only contains variables of one branch of the clock tree. As Section IV shows, this is an essential property, which ensures a deterministic evaluation of the expressions.

For example, consider the program that is shown in Figure 3. The module takes two inputs $a$ and $b$ and computes one output $gcd$, which is the greatest common divisor of the two inputs. Inside the module, a subclock $C_1$ is declared together with two local variables $x$ and $y$, which are related to this clock because they are declared in the scope of $C_1$. Then, an iteration computes the greatest common divisor, and the loop makes a step on the subclock each iteration as long as the result is not obtained. When the computation is finished, the output $gcd$ is set, and the module terminates. The delayed assignments to $x$ and $y$ take place at the next step of $C_1$, because both variables are declared for this clock. Hence, the values are updated for each loop iteration. The whole execution of the module needs some steps of the clock $C_1$ for the computation, but no step on the module clock.

### III. RELATED WORK

Using more than one clock in a system is a quite common approach to deal with timing problems including the ones mentioned in the introduction. For synchronous languages, many concepts and methodologies have been proposed that address the same problem. After we have shown our extension by subclocks in the previous section, we are now in the position to discuss its relationship to other approaches.

A multi-clock extension to Esterel is proposed in [6], which models multi-clocked systems by single-clocked Esterel. The interface of the system is extended by clock signals to trigger the different clock ticks. In contrast to our extension, the clocks must be provided by the environment. For determinism, the clocks must be generated deterministically, whereas our approach generates the clock ticks deterministically inside the system.

Multi-clocked systems can also be described by the synchronous language Lustre [13, 14]. Each Lustre program basically consists of a set of equations over data streams. In addition to functional symbols and delays, there are two operators to change the rate of a stream. The downsampling operator when takes a stream of arbitrary type and a Boolean stream and only keeps the events of the first one at those

---

Figure 1. Single-Clocked vs. Subclocked Systems

Figure 2. Clock Tree and Hourglass

Figure 3. Example: GCD

```
module GCD (nat ?a, ?b, !gcd)
{
    clock(C1) {
        int x = a, y = b;
        while(x > 0) {
            if(x >= y)
                next(x) = x - y;
            else
                next(y) = y - x;
            f1 : pause(C1);
        }
        gcd = y;
    }
}
```

---
instants where the second one is true. Upsampling is done by the current operator. This undoes a previous downsampling operation by inserting the last known value in the missing locations of the stream. Thus, all clocks in a Lustre program are subsets of the so-called base clock. Since upsampling only undoes the last downsampling, there is no means to refine the base clock. This difference to our clock-refinement approach is illustrated in Figure 2 (b). The base clock of Lustre 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 each of the sequentially ordered instants of the base clock, the clock calculus determines which values must be computed. Hence, it solves some of the language problems (maintenance, refinement) mentioned above, but the implementation restrictions still apply.

In contrast, the polychronous language Signal [11, 16, 17] can also cope with different clocks. While the syntax looks almost like Lustre, its semantics is very different due to its assumption that there is no base clock. As a consequence, Signal specifications are relational and not functional like Lustre: they do not describe a single behavior, but several ones, which differ in the clocks. Hence, Signal solves most of the above problems – 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 [22] or weakly endochronous [21]. 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 be safely 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 greatest common divisor (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. One can visualize the difference as follows: while the oversampling operator of Signal inserts new instants between existing ones, our subclocked model splits existing instants into new ones.

IV. Formal Semantics

Before we present a formal semantics of our language extension, we introduce some basic notations.

Definition 1 (Clocks): The set of all clocks which are defined in a synchronous program is denoted by \( C \), where \( C_0 \in C \) refers to the base clock. Since clocks are organized in a tree (according to their declaration), we write \( C_1 \succ C_2 \) if clock \( C_1 \) is on a higher level of the tree than clock \( C_2 \), and \( C_1 \succ C_2 \), if \( C_1 \succ C_2 \) or \( C_1 = C_2 \) holds.

Other operators such as \( \preceq \), \( \prec \), \( \preceq \), \( \not\preceq \), \( \not\prec \), and \( \not\preceq \) have their intuitive meaning. If two clocks are unrelated (neither \( C_1 \succ C_2 \) nor \( C_1 \preceq C_2 \) holds), we write \( C_1 \not\equiv C_2 \). For a statement \( S \), clocks \( (S) \) denotes the set of clocks which are declared in \( S \), e.g. clocks \( (\text{clock}(C) \{ S \}) = \{ C \} \cup (S) \). The supremum of two clocks \( C_1 \) and \( C_2 \) is defined by \( C := \sup(C_1, C_2) \) with \( C \succ C_1 \) and \( C \succ C_2 \) and \( \forall C' \in \mathbb{C} \), \( C' \prec C \) \( \Rightarrow \) \( (C' \not\equiv C_1) \lor (C' \not\equiv C_2) \). The supremum obviously exists since the clocks are organized in a tree.

Definition 2 (Variables): \( V \) is the set of variables of a synchronous program. Each variable \( x \in V \) stores a value of its domain \( \text{dom}(x) \), and it is declared in the scope of a clock, which is given by \( \text{clk}(x) \).

Definition 3 (Environment): An environment \( E \) maps each variable \( x \in V \) to a value of \( \text{dom}(x) \cup \{ \bot \} \). Hence, the extended domain of a variable \( x \) additionally contains the value \( \bot \), which is interpreted as not yet known. We write \( E(x) \) to retrieve the current value of \( x \) in environment \( E \), and similarly \( \[ \tau \]_E \) to evaluate expression \( \tau \) with respect to the values in environment \( E \). Evaluation of expressions is strict, i.e. if some part of it is evaluated to \( \bot \), then the whole expression is evaluated to \( \bot \). We write \( E_1 \subseteq E_2 \) for two environments iff \( \forall x \in V \). \( E_1(x) \neq \bot \rightarrow E_1(x) = E_2(x) \).

Definition 4 (Environment Restriction): A restriction of an environment \( E \) with respect to \( \odot C \) (where \( \odot \in \{ \succ, \prec, \preceq, \not\preceq, \not\prec \} \}) is defined as follows:

\[
(\mathcal{E})_{\odot C}(x) := \begin{cases} 
E(x) & \text{if } \text{clk}(x) \odot C \\
\bot & \text{otherwise}
\end{cases}
\]

Thus, \( (\mathcal{E})_{\not\odot C} \) 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 5 (Environment Combination): We define the intersection and union of two environments \( E_1 \) and \( E_2 \) formally by:

\[
(\mathcal{E}_1 \cap \mathcal{E}_2)(x) := \begin{cases} 
v & \text{if } v = \mathcal{E}_1(x) = \mathcal{E}_2(x) \\
\bot & \text{otherwise}
\end{cases}
\]

\[
(\mathcal{E}_1 \cup \mathcal{E}_2)(x) := \begin{cases} 
\mathcal{E}_1(x) & \text{if } \mathcal{E}_2(x) = \bot \\
\mathcal{E}_2(x) & \text{if } \mathcal{E}_1(x) = \bot \\
v & \text{if } v = \mathcal{E}_1(x) = \mathcal{E}_2(x)
\end{cases}
\]

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

In the following, we give a formal definition of the behavior of subclocked programs. We follow Plotkin’s structural approach to structural operational semantics (SOS) [19, 20], which defines the behavior by the derivation of rules over the program structure.

This approach has already been applied to synchronous languages [4, 5, 26, 29] in the past. Following [26], the semantics is divided into two parts, which can be modeled by two sets of rules: The transition rules determine for a given
environment and a program statement a residual statement, i.e. they model the transition from one program position to another and therefore describe the execution of a macro step. Since synchronous languages cannot be executed sequentially as given by the program structure, causality analysis [4] is usually needed to analyze the data dependencies within a macro step. Within the SOS approach, this can be modeled by a second set of reaction rules, which exactly describe the fixpoint iteration of causality analysis for the values and actions of the current macro step. Then, the overall behavior can be described by an alternating sequence of applications of the reaction and transition rules: first, the reaction rules are used to determine the current environment by means of micro step execution, and the determined environment is then used by the transition rules in order to execute a macro step of the program. The rules consider the syntax tree of a program. Thereby, the statements are either basic statements, or composed of others.

For our language extension, we reuse this basic structure. However, since we have steps of different granularity due to different clocks, rules are additionally parameterized by the clock. Instead of executing a single set of reaction rules, we simulate the next step for every clock by applying the reaction rules for each clock\(^2\). Similar to the single clocked approach, the reaction rules are applied in order to determine the maximum amount of information about activated actions which can be derived from the current situation without speculation. Then, we can decide for which clock the step must be performed and execute it by applying the transition rules for that clock. If this is the highest clock \(C_0\), inputs and outputs are additionally exchanged.

For imperative synchronous languages, a syntactically correct program is not necessarily semantically correct. There are typically two issues which must be considered: (1) write conflicts, which occur if different values are assigned to the same variable, and (2) causality problems, which are due to cyclic dependencies that result to an inconsistent or nondeterministic behavior or that cannot be resolved without speculation. Programs containing one of these problems are not desired and should be rejected, which should be also reflected in the semantics. Hence, the goal of the following definitions is two-fold. First, the semantics should specify whether a given program is valid at all. Second, for valid programs we want to determine the actual behavior. The following semantics rejects invalid programs by not providing a derivation with the given SOS rules, since their preconditions are not fulfilled. For valid programs, a derivation is possible, which also describes the behavior.

In Section IV-A, we first introduce the transition rules, while Section IV-B presents the reaction rules. The interaction between these two sets is defined in Section IV-C, before Section IV-D puts all parts together. This section also contains our main theorem about input/output determinism at the highest clock level.

### A. SOS Transition Rules

Although the reaction rules always precede the transition rules, we first consider them in this section. As already outlined above, the transition rules describe a step, which takes a given environment \(E\) and clock \(c\) and transforms a given statement \(S\) to a residual statement \(S'\), which is processed in the following step. Thus, they have the form:

\[
\langle E, S \rangle \xrightarrow{c} \langle S', c' \rangle
\]

Transitions are always made with respect to a given clock \(c\). Precondition for such a step is that such a step on \(c\) is possible and that the values of the variables by given environment match the assignments executed in the step. The transition rules return a residual statement \(S'\), which is considered in the following step, and a clock \(c' \in C \cup \{\ast\}\) which has been reached at the end of the step. If no concluding pause statement has been reached, we use the symbol \(\ast\) to indicate that \(S\) only consists of micro steps. Thereby, \(\ast\) is treated as a clock which is lower than all other clocks, i.e. \(\forall c \in C.\ c \succ \ast\) holds. We later need \(c'\) for the interaction of the rules. An invariant of the transition rules is that \((c' = \ast) \rightarrow (S' = \text{nothing})\) holds: if the statement \(S\) is executed as a micro step, nothing remains for the next step. This invariant can be straightforwardly checked for the rules.

Figure 4 shows the transition rules for our language extension\(^3\). The statements are considered recursively, thereby a statement is either a basic statement or consists of other statements. The rules \(a_1, a_2, p_1\), and \(p_2\) deal with basic statements, the remaining rules deal with combined statements. We first consider the transition rules for actions. Assignments are just micro steps, and therefore, the execution results in nothing as residual statement and \(\ast\) as clock (Rules \(a_1\) and \(a_2\)). The actual assignments are only considered by the reaction rules.

The behavior of the pause-statement depends on the two clocks parameters: The clock the \(\text{pause}(C)\) is defined for \((C)\) and the clock of the current step \((c)\). The \(\text{pause}(C)\) is consumed if the step-clock is the same as the pause-clock \((C = c, \text{Rule } p_2)\). If it is a lower one \((C \succ c, \text{Rule } p_1)\), the control flow will rest at this statement, and \(\text{pause}(C)\) will remain for the next step. There are two other possible cases for which no rules exist: (1) if \(C \prec c\) holds, then a lower pause statement is reached. However, this means that this step is not valid since the lower step (on \(C)\) should be executed. (2) If \(C \neq c\) holds, then an unrelated pause statement is reached. This is not possible since

\(^2\)As we will see later, the environment cannot be determined for all clocks in general, since higher clocks may still have to wait for the transition of lower clocks.

\(^3\)Due to lack of space, we only consider a subset of the Quartz statements that covers however all relevant aspects introduced by subclocks.
unrelated clocks can be either declared in other branches of a parallel statement, for which the transition rules of the parallel statements omit the execution of the other thread, or in a sequence of clock blocks, where the wrong clock would have been chosen. Hence, omitting these rules ensures that the transition fails in these cases, and the expected behavior of substeps is achieved.

Depending on the evaluation of the condition in the given environment, the conditional statement chooses one of its branches (Rules $c_1$ and $c_2$).

The rules for the $\text{clock}(C) \{S\}$ statement are straightforward: if the block $(S)$ is completely executed in the current step, and nothing remains for the following step, the whole statement will be removed (Rule $d_2$). Otherwise, the clock declaration is kept, and the residual statement remains for the next step (Rule $d_1$).

The first statement of a sequence is executed in every case. The execution of the second one depends on the result of the first: if the control flow rests inside the first statement (Rule $s_1$), the second one is not executed. If the first statement is executed as a micro step and the control flow does not rest inside (Rule $s_2$), the second one is also executed.

The execution of the parallel statement depends on the clock of the current step and the clocks which are declared inside the two threads. If the clock of the step is declared inside one of the threads, it cannot be visible for the other. Then, a step is done only for the corresponding thread (Rules $t_1$ and $t_2$), and the other thread is omitted. If the clock of the step is not declared inside one of the threads, it is a higher clock and therefore visible for both threads. In this case, the transition is done for both threads (Rule $t_3$). The resulting clock of the parallel statement is $\text{sup}(c_1, c_2)$ in this case. It basically means that both threads must terminate so that the whole parallel statement can terminate in the step.

B. SOS Reaction Rules

The reaction rules consider the data flow of the program in order to determine an environment for the current step so that a subsequent transition can be invoked. Since the clock of this transition is not known a priori, the reaction rules are applied for all clocks to simulate the next step for every clock. The reaction rules generally aim at finding out which assignments may and must be executed in the following step and which pause statements may or must be reached. Depending on this information, the semantics can check which clock steps are possible and then invoke a transition for any of them. The reaction rules operate on an incomplete environment, i.e. an environment which may assign $\bot$ to some variables. After each application of the rules this environment is updated on the basis of the information gained by the execution of the rules, before they are invoked again with the new information. A reaction rule has the following form:

$$\langle E', c, S \rangle \xrightarrow{c} \langle E', A^{\text{must}}, A^{\text{can}}, t, h \rangle$$

The reaction rules consider the current statement $S$ with the currently known information, which is encoded by the partial environment $E$. The clock for which the reaction rule is applied is denoted by $c$. In contrast, the clock $\hat{c}$ denotes
the current clock scope, i.e. the clock of the surrounding block of the statement which is currently considered. Thus, \( \bar{c} \) is updated when a clock block is entered. The result of the rules is a new environment \( \mathcal{E}' \), two sets \( \mathcal{A}_{\text{must}} \) and \( \mathcal{A}_{\text{can}} \) containing actions and two three-valued termination flags \( t \) and \( h \). Thereby, \( \mathcal{E}' \) is the resulting environment at the end of the statement, which is only used for the interaction of the reaction rules. The set \( \mathcal{A}_{\text{must}} \) contains the actions for which it is known that they are executed in the next step of the clock \( c \), while \( \mathcal{A}_{\text{can}} \) contains the actions which can be still executed although this is not yet exactly known due to incompleteness of the currently used environment. The value \( t \) indicates that the statement terminates in the next step of the clock \( c \), i.e. it is completely executed in the step. In addition, the value \( h \) is false if the statement does not terminate and the control flow rests at a higher clock than the considered one. \( t \) and \( h \) are three-valued variables, with \( t, h \in \{\text{true, false, } \bot\} \). Thereby, \( \bot \) means here also not yet known.

**Definition 6 (Ternary Operators):** The operators \( \land, \lor \) and \( \odot \) are defined by the following truth tables:

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>( \land )</td>
<td>( \lor )</td>
<td>( \odot )</td>
<td></td>
</tr>
</tbody>
</table>

The operators \( \land \) and \( \lor \) are the maximal ternary embeddings of their Boolean counterparts. The new operation \( \odot \) takes the value of its arguments if these are the same and is \( \bot \) otherwise.

The reaction rules are later used in a fixpoint iteration, which starts with an empty environment (i.e. the values of all variables are set to \( \bot \)) and then collects new information in the course of their derivation, which is used in the following iteration until no more changes occur in the environment.

The reaction rules are shown in Figure 5. Actions of the program are collected if the clock of the target variable of the action is higher or equal to the currently considered clock. One may assume that only the actions which belong to exactly the considered clock are collected. However, to decide whether a step on \( c \) can be performed or not, also assignments to variables of higher clocks must be considered, because they also happen in this step. Assignments to variables of lower clocks are ignored, because they may happen in a substep of \( c \), which cannot be decided here. However, the result is correct, because those actions are collected by the reaction rules for the lower clocks. The environment is not modified, the update of the actions will be committed after the application of the rules. The values \( t \) and \( h \) are both true, since actions are just micro steps, and they terminate in every case.

The behavior of a `pause` statement depends on the clock \( C \) for which the statement is defined and the considered clock \( c \). In every case, the resulting environment must be reset for variables which are declared for a clock lower or equal to \( C \). This is obvious, since after the `pause`, a new step starts for those variables, and their values are not yet known. If the considered clock is higher or equal to \( C \), \( t \) must be false (Rules \( p_1 \) and \( p_2 \)), because the control flow rests for the next step, and the statement does not terminate for the considered clock. \( h \) is set if \( c \) and \( C \) are equal. In the third case (Rule \( p_3 \)), the `pause` statement is defined for a lower clock. Those statements are considered as a micro step on a higher clock.

Another look at the Rule (\( p_1 \)) shows that it also handles unrelated clocks, i.e. \( C \neq c \) holds. The transition rules require that a step on the given clock is valid. So, we have to handle this case here in the reaction rules. However, an unrelated clock can be reached in two ways. (1) The parallel rule considers both branches. If we apply the reaction rules for a clock which is declared in one thread, the rule can encounter an unrelated clock in the other thread. (2) Clock blocks can appear in a sequence in the program, i.e. one clock is declared at the beginning of the program and one at the end. In this case, applying the rule for the later-declared clock may encounter a `pause` of the other one. However, if an unrelated clock is reached, we treat it as a higher clock, since a tick is not directly possible.

The rule for the clock block just updates the clock \( \bar{c} \). Therewith, the content of the block is considered with this clock, because it is in the scope of \( \bar{c} \). The other options are passed through.

The reaction rules for the conditional statement illustrate the meaning of the sets \( \mathcal{A}_{\text{must}} \) and \( \mathcal{A}_{\text{can}} \), similar to the transition rules, if the condition can be evaluated to true or false, one of the two branches is chosen (Rules (\( c_1 \)) and (\( c_2 \))). If the condition is evaluated to \( \bot \), which means that the value is not yet known, both branches are considered. The actions which can be executed in each branch are the actions which can be executed for the whole statement. On the other hand, only the action which must be executed in both branches, must also be executed for the whole statement. For the environment, we make use of the same approximation: only the values which are the same for both branches are taken. Nearly the same holds for \( t \) and \( h \). Only if both values are the same, this value is taken, otherwise it remains unknown.

The behavior of the sequence depends on the result of the reaction rules of the first statement. If the first statement does not terminate for the considered clock (Rule \( s_1 \)), i.e. \( t = \text{false} \), then the second one does not need to be considered because it is not reached in the next step for clock \( c \). If the first statement terminates for the considered clock (Rule \( s_2 \)), the second one is considered with the resulting environment \( \mathcal{E}_1 \) of the first one, because some variables which are declared for lower clocks can be reset.
during the first statement. The action sets are combined for both statements and the termination flags are used from the second statement. The third case considers an unknown termination for the first statement (Rule $s_3$). In this case, only the set $A_{\text{can}}^2$ from the second statement is used, because it is not known whether the statement is reached or not in the next step of clock $c$. The termination flags are also determined conservatively.

In contrast to the transition rules, there is just one reaction rule for the parallel statement. The Rule $t$ decomposes the statement and considers both threads separately. The result is then combined to get the result for the whole statement. Thereby, the most interesting parts are the termination flags.

The whole parallel statement only terminates if both threads terminate, i.e. $t_1 \land t_2$ holds. To explain the other flag, we consider the two cases for which $h = \text{true}$ must hold. (1) The whole statement terminates, then $t_1 \land t_2$ holds. (2) The control flow rests inside a pause statement of the considered clock. Then, $\neg t_1 \land h_1$ holds for the first thread or for the second accordingly. The disjunction combines the possible cases.

C. Clock Selection

The reaction rules of the previous section are applied for each clock declared in the program. The information retrieved by all these instances is now combined in order to
select the clock for the next transition. We first define the combination of all clocked reaction rules of the previous section to an unclocked version, which combines the reactions rules and removes the clock dependence.

**Definition 7 (Unclocked Reaction Rule):** Assume that for every clock \( c \in C \), there is a derivation of the form \( \langle E, \hat{c}, S \rangle \xrightarrow{\epsilon} \langle E', A_c^{\must}, A_c^{\can}, t_c, h_c \rangle \). Then, the corresponding unclocked reaction rule is written as

\[
\langle E, \hat{c}, S \rangle \xrightarrow{\epsilon} \langle A^{\must}, A^{\can}, C^{\en} \rangle
\]

where

- \( A^{\must} = \bigcup_{c \in C} A^{\must}_c \)
- \( A^{\can} = \bigcup_{c \in C} A^{\can}_c \)
- \( C^{\def} = \{ c \in C \mid h_c = \text{true} \land \forall c' \in C, c' < c \rightarrow h_{c'} = \text{false} \} \land \{ c \in C \mid \forall c' \in C, c' < c \rightarrow \forall v \in V, \text{clk} (c) \leq c \rightarrow \text{clk} (c') \leq c \land \forall (\hat{c}, \alpha, v) \in A^{\can}_c \}
- \( C^{\en} = C^{\def} \cap \bigcap_{c \in C} C^{\can}_c \)
- \( \forall (c, x, v) \in A^{\must}, (E(x) \neq \bot \land v \neq \bot) \rightarrow E(x) = v \)

The action sets are obtained by combining the sets of all clocks. \( C^{\en} \) is the set of clocks which are enabled, i.e., for which the next step can be performed. Therefore, \( C^{\en} \) consists of the clocks, which are enabled by the control flow \( (C^{\def}) \) and by the data flow \( (C^{\can}) \). A clock \( c \) is contained in \( C^{\def} \) if a \texttt{pause} statement of this clock is reachable and all lower clocks reach the same \texttt{pause}. In detail, the lower clocks must reach a higher \texttt{pause} and therefore \( h_{c'} = \text{false} \) must hold for all lower clocks \( c' \). The additional condition, which checks \( t_{c'} = \text{true} \) for the lower clocks, handles the case when no \texttt{pause} statement is reached, i.e., the end of the module is reached and all steps are micro steps. Then, the highest (module) clock is enabled\(^4\). A clock \( c \) is contained in \( C^{\can} \) if all variables for \( c \) and for all lower clocks are known and all collected actions which are executed by the step can be evaluated by the environment, i.e., all data dependencies are resolved. The last condition ensures that no \texttt{write-conflicts} occur, i.e., no variable is assigned with different values in the same step. Therefore, every assigned value must be consistent to the value stored by the environment. If this is not the case, the rule is not applicable and the program has no valid execution.

**Lemma 8 (Invariants):** Given an unclocked reaction rule \( \langle E, \hat{c}, S \rangle \xrightarrow{\epsilon} \langle A^{\must}, A^{\can}, C^{\en} \rangle \), then the following conditions hold:

- \( A^{\must} \subseteq A^{\can} \)
- \( \forall c_1, c_2 \in C^{\en}, c_1 \neq c_2 \)

**Proof:** The first property is satisfied by every clocked SOS reaction rule given in Figure 5 and it is preserved by the union of the sets for all clocks. The second property ensures that only unrelated clocks are enabled at the same time. One can verify that it is fulfilled by inspecting the termination flags in the reaction rules and the definition of \( C^{\en} \). The only crucial case is imposed by the parallel statements, where a subblock is defined for each thread, and the threads can make independent steps.

**Lemma 9 (Reaction Rule Monotonicity):** The application of rules is monotonic, i.e., for two environments \( E_1 \subseteq E_2 \):

\[
\langle E_1, \hat{c}, S \rangle \xrightarrow{\epsilon} \langle A_1^{\must}, A_1^{\can}, C_1^{\en} \rangle \}
\langle E_2, \hat{c}, S \rangle \xrightarrow{\epsilon} \langle A_2^{\must}, A_2^{\can}, C_2^{\en} \rangle
\]

the following holds:

- \( A_1^{\must} \subseteq A_2^{\must} \)
- \( A_1^{\can} \subseteq A_2^{\can} \)
- \( C_1^{\en} \subseteq C_2^{\en} \)

**Proof:** This is also directly satisfied by the definition of the reaction rules. If the information given by the environment increases, i.e., more variables are defined, the information derived by the reaction rules also increases.

The environment and the clock to perform a step is derived by iteratively applying the reaction rules. The iteration is stopped if no more changes occur in the environment, i.e., the maximal possible information is derived. This fixpoint iteration is covered by the following definition.

**Definition 10 (Environment Fixpoint):** Let \( (E_1, E_2, \ldots, E_n) \) be environments such that for all \( 1 \leq i < n \), derivations of the form \( \langle E_i, C_0, S \rangle \xrightarrow{\epsilon} \langle A_i^{\must}, A_i^{\can}, C_i^{\en} \rangle \) exist that satisfy the following conditions:

1. \( E_1 = \langle E \rangle \xrightarrow{\epsilon} (D)_{\epsilon \in \epsilon} \)
2. \( E_{i+1} (x) = \begin{cases} E(x) & \text{if } (\hat{c}, x, v) \in A^{\must}_i \\ E_i (x) & \text{otherwise} \end{cases} \)
3. \( E' = E_{n-1} = E_n \)
4. \( D'(x) = \begin{cases} D(x) & \text{if } \text{clk} (x) = c \\ v & \text{if } (\hat{c}, \text{next }, (x), v) \in A^{\can}_{n-1} \\ \bot & \text{otherwise} \end{cases} \)
5. \( c' \in C_{n-1} \)

Then, we write

\[
\langle E, S, D, c \rangle \xrightarrow{\epsilon} \langle E', D', c' \rangle
\]

Thereby, \( E \) is the environment of the last step, \( S \) is the current statement to consider and \( c \) is the clock of the last step. \( D \) contains the values of variables assigned by delayed action from the last step. The fixpoint iteration determines the environment \( E' \) and the clock \( c' \) for the current step.

Values of variables of the last step do not have to be necessarily valid for the current step. This depends on the clock the last step was performed for. Therefore, all variables belonging to this clock \( c \) are reset to \( \bot \) for the iteration (1). Values of other variables are kept. The reaction rules are applied with this environment and according to the action.

\(^4\)When the end is reached, the system stutters at the highest level.
sets the environment is updated (2). A variable is set to a value if there is an action in \( A^{\text{must}} \) and the expression can be evaluated to a value. The value from the previous step is taken if no action can set this variable in the next step of the clock \( \text{clk}(x) \). (3) ensures that the fixpoint is reached. The delayed assignments are transferred by \( D' \) to the next step (4). Thereby, delayed assignments from the last step must be preserved if they did not take place in the current step because of the clock \( c \). The clock for the next step is chosen from the enabled clocks after the last iteration (5). This is a nondeterministic choice.

**Lemma 11 (Environment Monotonicity):** The sequence of environments \((E_1, E_2, \ldots, E_n)\) used in Definition 10 is monotonic, i.e., \( E_1 \subseteq E_2 \subseteq \ldots \subseteq E_n \). The sequence starts with \( E_1 \). The other environments are each obtained by setting more values. Due to the absence of write-conflicts, which is ensured by Definition 7, no conflicting values can be set. Thus, values of variables can only change from \( \bot \) to another value. The monotonicity of Lemma 11 ensures that the fixpoint iteration can be efficiently computed according to the Tarski-Knaster fixpoint theorem [28]: starting with the least value and then reapplying the reaction rules until a fixpoint is reached.

The existence of the fixpoint is ensured by a finite number of variables: the number of variables are the upper bound which assigns \( E \) to the next step. Hence, in every step, we first determine the environment and then execute the program with the clock of the step, and then execute the program with the highest clock.

**Definition 12 (Step):** A step of a subclocked synchronous system is defined by \( (E, S, D, c) \mapsto (E', S', D', c') \) with

- \( (E, S, D, c) \mapsto (E', D', c') \) (determine fixpoint)
- \( (E', S) \xrightarrow{c} (S', c') \) (perform transition)

Hence, in every step, we first determine the environment and the clock of the step, and then execute the program with the computed environment and step. The nature of subclocks is well described by this definition: before starting a step, it is not determined which clock the step is for. Thus, subclocks can be seen as an internal logical refinement and the clock signals cannot be provided from the outside of the system.

**Theorem 13 (Change Consistency):** For a step on a clock \( c \), the environment only changes for variables of this clock:

\[
( (E, S, D, c) \mapsto (E', S', D', c') ) \rightarrow \left( (E)_{c \neq 1} \subseteq (E')_{c \neq 1} \right)
\]

**Proof:** Variables can be only set from \( \bot \) to a value. From the definitions, it can be immediately seen that values only change for variables declared on \( c \) or lower clocks. The choice of the clock of a step can be nondeterministic if there are parallel threads which can perform independent substeps, which results to two unrelated clocks that are enabled for the next step. However, we want to ensure deterministic behavior for the whole program, i.e. at the highest clock. Therefore, independent choices must not influence each other.

**Theorem 14 (Procrastination):** Given the current state of the execution of a subclocked program with \( E, S \) and the clock of the last step \( c \). Assume that two next steps are possible:

\[
(E, S, D, c) \mapsto (E_1, S_1, D_1, c_1)
\]

Then, there exists \( E_2', S_2', D_2' \) with:

\[
(E_1, S_1, D_1, c_1) \mapsto (E_2', S_2', D_2', c_2)
\]

Hence, we have the often desired diamond property: if two different clocks \( c_1, c_2 \) are enabled for the next step, then doing the step on \( c_1 \) preserves that \( c_2 \) is still enabled. Furthermore, if \( c_2 \) catches up, results will be the same for variables on the clock \( c_2 \).

**Proof:** Due to Lemma 8 \( c_1 \neq c_2 \) must hold. Therefore, variables affecting the step of \( c_2 \) are not modified by \( c_1 \), i.e. \( (E)_{c \neq 1} \subseteq (E_1)_{c \neq 1} \) (Theorem 13). Thus, for \( c_2 \) a greater environment is available, because the variables which can change are not visible for \( c_2 \). The variables visible for \( c_2 \) can just change from \( \bot \) to a value. Due to the monotonicity and the fact that a transition for \( c_1 \) does not affect the clock scope of \( c_2 \), \( c_2 \) is still enabled and the step results in the same values, i.e. \( (E_2)_{c \neq 1} = (E_2')_{c \neq 1} \).

Combining Theorem 13 and Theorem 14, we get our desired result that subclocked synchronous programs are deterministic at the highest clock.

V. Example

In this section, we give a small example which illustrates how the rules are applied. Figure 6 (a) shows a sample program where two threads run in parallel and a subclock is defined for each thread. The threads are shown beside each other for space reasons and easier understanding. The two clocks \( C_1 \) and \( C_2 \) can make independent steps, as long as there are no data dependencies between them. The clock tree of the program is shown in Figure 6 (b). The variables of the module interface \( x \) and \( o \), and the local variables \( y \) and \( z \) are declared on the highest clock \( C_0 \). The variable \( c \) is declared for \( C_1 \) and no variables are declared for \( C_2 \). We start with the initial environment \( E^{\text{init}} \) which assigns true to the input \( x \) for the first step. All other values are unknown and thus, they are evaluated to \( \bot \).
We implemented a simulator based on the presented SOS rules. For the example program, Figure 6 (c) shows a snapshot from the last cycle of the fixpoint iteration for the first step. It provides the sets $A^{\text{must}}$ and $A^{\text{can}}$ as well as the enabled clocks in detail. Obviously, the actions which set $z$ and $c$ are contained in $A^{\text{must}}$. The action which sets $y$ is also contained, but the right-hand-side value is unknown, since in order to reach this assignment in the source code a pause statement on clock $C_1$ must be passed. The same actions contained in $A^{\text{must}}$ are also contained in $A^{\text{can}}$. In addition, the delayed assignment of $o$ is only contained in $A^{\text{can}}$ because the execution of it is guarded by the if ($y$) whose condition is unknown ($\perp$). From the control-flow side, both subblocks $C_1$ and $C_2$ are enabled and therefore, no higher one can be enabled. From the data-flow side, only $C_1$ is enabled, because all actions which must be executed on a step of $C_1$ can be evaluated, and the values of all variables declared for $C_1$ are known. For $C_2$ this is not the case, because it is not known whether the delayed assignment for $o$ must be executed or not. Hence, the only possible clock tick is $C_1$. The derived environment assigns the derived values to the variables, only $y$ is assigned by $\perp$. For the output $o$, no action remains in the set $A^{\text{can}}$ which could assign it to a value in the current step (of $C_0$). Thus, it is set to its default value 0.

Applying the transition rules with the resulting environment and the only possible clock for the next step ($C_1$) results in the new statement shown in Figure 6 (d). Since the clock $C_1$ is declared inside the first thread of the parallel statement, only this thread is affected by the program transition. For the following step, the variables of $C_1$ are reset since they may get a new value for the following step. All other values remain the same.

The example shows some aspects which are recalled here. If it is not known that an assignment is executed or not, it is contained in the set $A^{\text{can}}$ but not in $A^{\text{must}}$. If the value which must be assigned is not known, it remains $\perp$. However, both cases can prevent a clock from being enabled.

$A^{\text{must}}$ also contains the assignment to the variable $y$. It stems from the application of the reaction rules for the clock $C_0$. The variable $z$ has still the value false. When the rules reach the pause statement at $\ell_{11}$ it just steps over because the control flow of $C_0$ cannot rest here. However, some variables (just $c$ for the example) in the environment must be reset, because they are no longer valid in the following step of $C_1$. Therefore, the value of $z$ remains and the if ($z$) can be evaluated. It is sure that the assignment to $y$ is reached, but the right-hand-side value is not known, because this is not yet known for this substep.

VI. CONCLUSIONS

We introduced the refinement of clocks into subblocks to overcome certain limitations of the single-clocked synchronous model of computation. To this end, we propose new statements for the synchronous language Quartz and presented a new structural operational semantics with subclocks. Moreover, we proved that the obtained synchronous programs with subclocks still generate unique output sequences for given input sequences even though subclocked statements with unrelated clocks can be asynchronously executed and only synchronize at their least common clock.

Our extension tackles the problems mentioned in the introduction: The over-synchronization, i.e. synchronization of parallel threads where it is not necessary, can now be avoided since the developer is given the ability to describe independent steps of parallel threads and to explicitly synchronize the threads on some higher clock level.
Furthermore, the conceptual basis for sequential functions in synchronous languages are given. As it is shown by the example which calculates the greatest common divisor (Figure 3), computation results can be obtained in the same step, even if the actual computation needs several (invisible) substeps.

The semantics itself remains complex because it covers two aspects: (1) it determines whether a program is semantically correct or not, and (2) for semantically correct programs, it determines their behavior. Synthesis procedures for our subclocked language do not have to deal with this complexity since they can rely on the fact that the validity of programs has already been ensured by a static analysis in advance.

References


