Causality Analysis of Synchronous Programs with Refined Clocks

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

Abstract—Synchronous languages are based on the synchronous abstraction of time, which divides the execution of programs into an infinite sequence of micro steps that consist of finitely many macro steps. A well-studied problem of this model of computation are cyclic dependencies of micro steps whose constructiveness has to be checked by a causality analysis during compilation.

Recently, we showed that temporal refinement can be introduced to imperative synchronous languages by refined clocks. In this paper, we formally define the causality analysis for this extension. To this end, we translate the program into a transition system, which can then be used to verify the correct causal behavior with a model checker. We also list optimizations that can be used by compilers to conservatively approximate causality checking.

I. INTRODUCTION

The design of efficient and reliable concurrent systems is an inherently difficult task. For developers it is often difficult or impossible to determine which states a particular set of components may reach or what events in a system may happen simultaneously. The consequence of this complexity is the design of systems that show unpredicted or unwanted behavior such as deadlocks or data races.

The synchronous abstraction of time [2, 3] tries to avoid some part of the complexity by introducing a logical timescale, which divides the execution of a system into a sequence of reactions (sometimes called instants or macro-steps). Within a macro step, time is abstracted, and all the events belonging to it are assumed to happen synchronously. As all threads of a system are based on the same scale, they all run in lockstep, i.e. they synchronize in each step as known from barrier synchronizations. The abstraction to macro steps reduces the complexity of the model, as it reduces the number of reachable states of a system.

This core idea has made the application of synchronous system design very successful in many areas. In hardware design [2], almost only synchronous circuits are implemented, and most automatic verification tools based on symbolic model checking consider synchronous systems. The race-free deterministic concurrency also proved to be very suitable for embedded systems software: parallel systems can be better statically analyzed and many systems can be even statically scheduled, which makes expensive dynamic thread management superfluous.

On the other hand, the temporal abstraction has some challenging consequences for compilers and code generators. While all the computation and communication within an instant does not take time in theory, this is not true in reality. To ensure the correctness of the synchronous abstraction of time, (1) the execution of micro steps follows the data dependencies (so that everything seems to be executed simultaneously), and that (2) the instants where macro steps are executed have to be sufficiently long so that the system can stabilize in between. In synchronous hardware circuits, which make use of the synchronous abstraction of time, the clock is responsible for the second property, while the first one must be ensured by the signal propagation of gates.

Related Work: As known from hardware circuits, combinational cycles in synchronous circuits might cause problems [22, 23]. Therefore, causality analysis has been an important research topic in the context of synchronous programming languages [19] such as Esterel [4, 5], Lustre [15], Signal [10, 17] or Quartz [18]. In contrast to hardware circuits, which are often required to be acyclic in practice, these languages allow cycles provided that the execution can resolve the cycles in a constructive way.

In order to define this, the notion of constructive programs was introduced in this area. Such programs do not only have a single well-defined consistent behavior but they also guarantee that this behavior can be constructively determined without any guessing. The notion of constructive programs is closely related to intuitionistic/constructive logic [5]. Furthermore, it can be shown that a causally correct synchronous program can be translated to a hardware circuit which stabilizes for arbitrary gate delays [8] or to a set of tasks which can be always dynamically scheduled according to their data dependencies [9].

Contribution: In this paper, we generalize previous work on causality analysis of synchronous languages. Whereas previous work relies on a single clock, i.e. logical time is divided into a sequence of equal instants, our model of computation is more general: it allows the refinement of steps into substeps so that we have a hierarchical partition of the program execution. Obviously, this requires a more difficult causality analysis as events on different levels must be analyzed.

Our contribution is twofold: (1) we give a formal definition of causality for systems with refined clocks, and (2)
we list conservative approximations to quickly check the causality of a system at compile-time.

Outline: The rest of the paper is structured as follows: Section II presents the basic notions of imperative synchronous programs and introduces the concept of clock refinement. Section III presents the first part of our contribution: we define causality in the context of clock refinement, and reduce it to a model checking problem. Section IV contains the second part of our contribution: conservative approximations which can be used by compilers to efficiently check causality. Finally, Section V concludes with a short summary.

II. CLOCK REFINEMENT

A. Clock Refinement in Imperative Synchronous Languages

The synchronous model of computation [3, 14] divides the execution of a program into a sequence of macro steps [16] (also called reactions). In each of these steps, the system reads the inputs, performs some computation and finally produces the outputs. In theory, the outputs appear the same instant when the inputs are read. In practice, the execution implicitly follows the data dependencies between the micro steps, and outputs are 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, a pause marks the end of a reaction and is therefore responsible for consuming one unit of logical time. Thus, a reaction consists of all actions (or assignments) between two consecutive pause statements.

As already stated in the introduction, all threads are based on the same timescale and therefore, synchronize at each pause statement – even if they do not communicate. This so-called over-synchronization is an undesired side-effect of the synchronous model of computation.

While many solutions exist to overcome this problem (clock sampling in Lustre, polychrony in Signal, multi-clock extensions in Esterel), we focus on clock refinement in the following. This extension was recently proposed [12] to avoid over-synchronization and other undesired effects in synchronous languages. Its basic idea will be illustrated in the following 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 clock refinement. The module reads its two inputs a and b in the first step and assigns them to the local variables x and y. Then, the module computes iteratively the GCD of the local variables. The computation steps are separated by the pause statement with label ℓ. Each variable has one value per step and the delayed assignments in form of the next statements assigns a new value to the variables for the following step. Finally, the

```
module GCD1
(nat ?a, ?b, !gcd)
{
    nat x, y;
    x = a;
    y = b;
    while(x > 0) {
        if(x >= y) {
            next(x) = x-y;
        } else {
            next(y) = y-x;
        }
        ℓ: pause;
    }
    gcd = y;
}
```

(a) Single Clock

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

(b) Clock Refinement

Figure 1. Greatest Common Divisor

GCD is written to the output gcd. Apparently, a drawback of this implementation is that the computation is spread over a number of reactions whose number depends on the input values, and each call to this module has to take care of the consumption of time.

The second variant, shown on the right-hand side of Figure 1, makes use of clock refinement. While the overall algorithm is the same, the time required for the GCD computation is now hidden by 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 a pause statement of the outer clock and thus, the computation steps are not visible to the outside. As a consequence, each call to this module seems to be completed in a single step. The local variables x and y are now declared inside the local clock block and therefore, they can change their value for each step of the local clock, which is crucial for the correctness of the algorithm.

With local clock declarations, existing clocks can be refined and arbitrary many abstraction levels can be introduced to the single-clocked synchronous model. Furthermore, the extension gives developers some flexibility for desynchronized implementations through unrelated clock refinements, i.e. local clocks that are declared in parallel threads. Those clocks are not visible to each other, and both threads can proceed independent of each other until a synchronization is enforced by a pause of a common clock. Thus, the clock declarations generally form a tree, and clocks are not visible outside their local declarations.

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 so that 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 \( C_0 \) 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 \( \succ \), \( \prec \), and \( \preceq \) are used accordingly. If two clocks \( C_1 \) and \( C_2 \) are independent, i.e. neither \( C_1 \succ C_2 \) nor \( C_1 \prec C_2 \) holds, we write \( C_1 \not\preceq C_2 \). This may hold if the two clocks are declared in different threads of a parallel statement.

\[ \begin{aligned}
\text{module } \text{Causality1} & \quad \text{module } \text{Causality2} \\
(\text{nat } ?a, x, y) & \quad (\text{nat } ?a, x, y) \\
\{ & \\
\ell_1 : \text{pause}; & \ell_1 : \text{pause}; \\
y = a + x; & \text{if } (y > 2) \\
x = 2 * a; & x = 2 * a; \\
\ell_2 : \text{pause}; & \ell_2 : \text{pause}; \\
\} & | \\
& \quad \{ \\
y = a + 1; & \}
\end{aligned} \]

Figure 2. Causality in the Synchronous Model

\[ \begin{aligned}
\text{module } \text{Causality3} (\text{nat } ?a, o) \\
\{ & \\
nat c; & \\
\ell_0 : \text{pause}; & \\
\text{clock}(C1) \{ & \\
\quad \text{nat } x = a; & \\
\quad \ell_1 : \text{pause}(C1); & \\
\quad \text{if } (x > 2) & \\
\quad \quad o = c; & \\
\quad \text{else} & \\
\quad \quad c = 4; & \\
\quad \ell_3 : \text{pause}; & \\
\} & | \\
& \quad \{ \\
\quad \text{clock}(C2) \{ & \\
\quad \text{nat } y = a; & \\
\quad \ell_2 : \text{pause}(C2); & \\
\quad \text{if } (y \leq 2) & \\
\quad \quad o = c; & \\
\quad \text{else} & \\
\quad \quad c = 3; & \\
\quad \ell_4 : \text{pause}; & \\
\} & \}
\end{aligned} \]

Figure 3. Causality Example with Refined Clocks

\( \ell_4 \). Thus, if one of the threads depend on this variable, it cannot proceed until the value is known. The variable \( c \) is read and written by both threads, so that actually a cyclic dependency arises. However, it can be easily seen by the condition of the \texttt{if} statement that only one direction of this dependency can be present, so that there is no problem for the execution of this example. If both dependencies would be present, a deadlock would occur: no thread can proceed due to unresolved variable values. At the end, the execution needs to be scheduled accordingly, if this is possible. In the following, we will focus on the question, if such a schedule exists, or if the system may run into a deadlock.
existence of such schedules are considered in this paper.

The first kind of causality can be dynamically resolved by single execution step but is spread over several substeps. For clock refinement, the intermediate representation as well as the original compilation algorithm were recently extended to cover the additional temporal layers [11, 13]. In the following, we briefly introduce this representation since it forms the starting point of our analysis. Thereby, we do not have to deal with the rich set of Quartz statements and their semantic challenges. Instead, we can focus on the core of the problem, i.e., the analysis of causal dependencies.

As already stated above, AIF uses guarded actions to represent both the data flow and also the control flow of the system. Thereby, each guarded action is of one of the following forms:

\[
\gamma \Rightarrow x = \tau \\
\gamma \Rightarrow next(x) = \tau
\]

The Boolean condition \(\gamma\) is called the guard of the action. If it holds in a given reaction, the action is executed. Thereby, the immediate action \(x = \tau\) assigns the value of the expression \(\tau\) to the variable \(x\) in the same reaction. In contrast, the delayed action \(next(x) = \tau\) only evaluates the value of the expression \(\tau\) and assigns the value in the next step of the clock \(clock(x)\) (the next time \(x\) can change). Thus, the assignment is visible in the next execution step that belongs to the clock of the variable \(x\). Due to the synchronous semantics, all guarded actions are evaluated at once and immediate variable changes are directly visible. Causality means in this context that the actions are executed according to their data dependencies. The clocks of each variable and the relation of the different clocks are also stored in the intermediate format.

The guarded actions of the greatest common divisor in Figure 1 (b) are shown in Figure 4. The special symbol \(st\) is the start signal of the module, which is expected to hold exactly in its first step. The actions in the loop can be executed when either the module is started and the loop is entered, or when the control flow is at the \(pause\) statement with label \(\ell\) and the loop is restarted. The variable \(qrz\) is assigned when the loop condition does not hold. When the control flow is at a label, it can only go ahead, when the clock variable of this label holds. Therefore, all labels occur in combination with their clock variable. This seems to be redundant, but it is not, because the labels store the current control-flow state of the module, whereas the clock just holds for a single execution step. In parallel threads, one thread can proceed by a clock that is not known by the other thread, while the other one stays at its current label.

### B. Transition System

As explained above, causality analysis checks whether there is schedule in each step to constructively determine all the variables. Hence, it is crucial to know which variables have already been determined. Therefore, causality analysis is based on a model which explicitly stores whether the value of variable is known at a particular instant or not. In the

<table>
<thead>
<tr>
<th>Figure 4. Guarded Actions of GCD Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(st \land C0 \Rightarrow x = a)</td>
</tr>
<tr>
<td>(st \land C0 \Rightarrow y = b)</td>
</tr>
<tr>
<td>(st \land C0 \land x &gt; 0 \land x \geq y \Rightarrow next(x) = x - y)</td>
</tr>
<tr>
<td>(st \land C0 \land x &gt; 0 \land \neg(x \geq y) \Rightarrow next(y) = y - x)</td>
</tr>
<tr>
<td>(st \land C0 \land x &gt; 0 \Rightarrow next(\ell) = true)</td>
</tr>
<tr>
<td>(st \land C0 \land \neg(x &gt; 0) \Rightarrow gcd = y)</td>
</tr>
<tr>
<td>(\ell \land C1 \land x &gt; 0 \land x \geq y \Rightarrow next(x) = x - y)</td>
</tr>
<tr>
<td>(\ell \land C1 \land x &gt; 0 \land \neg(x \geq y) \Rightarrow next(y) = y - x)</td>
</tr>
<tr>
<td>(\ell \land C1 \land \neg(x &gt; 0) \Rightarrow gcd = y)</td>
</tr>
</tbody>
</table>

To summarize, causality of a single execution step is considered in synchronous languages for a long time, and it is a well-studied problem. With the extension of the synchronous model with refined clocks, a new kind of causality has been introduced which does no longer take place in a synchronous model with refined clocks, a new kind of causality can be obtained by compiling Quartz files as described in [7, 18, 20]. AIF (Averest Intermediate Format), which essentially consist of a set of synchronous guarded actions. The intermediate representation can be obtained by compiling Quartz files as described in [7, 18, 20]. AIF is a good choice for further analysis since it abstracts from the complexity of the source language by resolving the difficult interaction of statements. Nevertheless, AIF files still contain the entire behavior (including its causal properties) of the given synchronous program.

\(^1\)http://www.averest.org
following, we define a transition system which is not only defined over the variables of the systems but additionally over a set of known-flags (one for each variable written by the system) similar to [6, 19]. These flags denote the status of each variable and can be used to check whether all expressions can already be evaluated in a current instant so that a thread can proceed or whether it has to wait until another one provides the necessary information (e.g. consider the program in Figure 3). By taking the transition system of the plain system model and cutting off transitions without sufficient information, causality problems lead to stuttering in dead-end states, which can be detected by a model checker.

For the definition of the transition system, we reuse this basic idea from the single-clocked case, where causality has already been reduced to a model checking problem [19] in a similar way. However, clock refinement and the resulting substeps impose additional problems which must be addressed in the analysis. In addition to cyclic dependencies in a single instant, cycles may span over several smaller steps which are combined in a larger step on a higher level. At the end, it must be checked whether substeps can always be scheduled according to their data dependencies.

Due to the lack of space, we cannot describe all the details of the transition system in this paper, but focus on the general principle and the crucial parts that are required in the following section.

For the definition of the known-flags, consider a part of an execution trace. A variable $x$ has a single value once it becomes known, i.e. between two consecutive ticks of its clock $\text{clock}(x)$, which also holds if a higher clock ticks. Therefore, we can simply distinguish between ticks of clock $(x)$ and all lower clocks. A single path for the variable $x$ is shown in Figure 5. The small dots are instants in which $\text{clock}(x)$ does not hold. There, a lower clock, an unrelated clock or even no clock may hold, whereas the big dots are instants in which $\text{clock}(x)$ holds. However, assignments to $x$ can also be done from lower clocks levels, i.e. in the small instants. The example shows a delayed assignment that is evaluated directly, but the value for $x$ is updated at the next instant when $\text{clock}(x)$ holds. There is also an immediate assignment shown that directly updates the value of $x$. As already said, a variable has exactly one value per step (i.e. between the ticks of $\text{clock}(x)$). However, the value may not be known at the beginning since the variable can be set after some smaller substeps. Obviously, the variable can only be used, once its value is known. An immediate assignment makes a variable known for the remainder of the step, i.e. until the next instant where $\text{clock}(x)$ holds. A delayed assignment makes a variable known for the following step. Both possibilities are illustrated Figure 5. In addition, a variable is also known, when no assignment is performed to it in a step, then it remains unchanged and keeps its value from the previous step.

With this preliminary considerations, we can now define the property known $(x)$ for a variable $x$. Therefore, assume that the following guarded actions exists in a system writing $x$:

$$\gamma_1 \Rightarrow x = \tau_1 \quad \chi_1 \Rightarrow \text{next}(x) = \pi_1$$
$$\vdots$$
$$\gamma_p \Rightarrow x = \tau_p \quad \chi_q \Rightarrow \text{next}(x) = \pi_q$$

The variable $x$ is then known, when it is set by an immediate assignment in the current step, when it was set by a delayed assignment from the previous step or when it will not be written in the current step. Assume that the term $\text{set}_x$ covers the first two cases where $x$ have been set by a delayed assignment or by an immediate one for the current step. Then, we can define by using the strong until-operator $[\_ \bigcup \_]$ of linear temporal logic:

$$\text{known } (x) \equiv \text{set}_x \lor \text{next} \left( \left( \bigwedge_{i=1}^{p} \neg \gamma_i \bigcup \text{clock}(x) \right) \right)$$

For the rest of the paper, the second part of this expression is the more interesting one. It expresses that the variable $x$ can also be used when it is not set, but it is also known that it will not be set in this step at all.

IV. CONSERVATIVE APPROXIMATIONS

When considering the complexity of a transition system for the whole system enriched by additional flags (as the one proposed in the previous section), one soon experiences that this kind of causality checking does not scale well due to the enormous state-space explosion. As the model checker needs to build the entire state space, only small systems (or system components) can be checked this way. This problem is already known from traditional causality analysis of synchronous systems, but becomes even more problematic with refined clocks.

For this reason, we develop conservative causality checks, which might refuse some causally correct programs but which are simple enough to be automatically checked at compile-time. For the single clocked case, this approximation often consists in checking whether the system is acyclic, i.e. whether the graph of immediate dependencies
does not contain cycles. In the following, we will show that similar checks exist for our more general case with refined clocks. The precise definition based on the transition will serve as a correction criterion: all programs accepted by the following simple checks, are causally correct with respect to this definition made in the previous section.

A. Acyclic Clock Levels

In the transition system, the definition of the expression known\( (x) \) is rather difficult: it involves an until-operator, which models that if \( x \) has not been assigned in its current step, it is only known if it cannot be assigned in a later substep. Thus, later substeps have an influence on the current one since they are linked by \( x \).

This complex expression can be avoided by forbidding immediate assignments to variables of slower clocks. Then, only delayed assignments to such variables are allowed and the following simplification holds

\[
\text{next} \left( \bigwedge_{i=1}^{p} \neg \gamma_i \cup \text{clock} (x) \right) \equiv \text{true}
\]

As a consequence, information only flows from slower levels to higher levels in the same step\(^2\). Thereby, all causality problems introduced by our extension are eliminated since concurrent threads in different clock scopes only exchange data at the beginning of common steps. In addition to causality, this property also ensures the maximum amount of parallelism between unrelated clock scopes: they only need to synchronize at common pause statements, while they are completely independent for the rest of the time.

B. Acyclic Information Flow through Clock Levels

Refusing programs with immediate assignments to variables on higher clocks is a simple, but an unnecessarily strict condition. For instance, reconsider the introductory example in Figure 1 (b) which computes the greatest common divisor. There, the variable \( \text{gcd} \) is set by such an immediate assignment, where \( \text{gcd} \) is defined on a higher clock level. We could use a delayed assignment instead, but then, the result of the computation would be only available in the following step, which does not allow us to construct modules such as \( \text{GCD2} \) to compute functions.

If we have another look at this example, we can see that the variable \( \text{gcd} \) is always assigned at the end of the module. However, this property can only be checked with the help of a reachability analysis, which is not better than the complete check based on the transition system.

We also see in the example that the execution of the subclock scope does not depend on the variable \( \text{gcd} \), because it is only written but not read. Thus, if each clock block either reads or writes a variable on a higher clock, the writing blocks can be executed before the reading blocks and after all writing blocks are finished, the variable is known. This is either the case because the variable was written by a block or no block has written it. In this second case, it is also known that no other block will write it.

Hence, for the expression known\( (x) \) we can achieve the same simplification as in our first approximation. Since the substeps that write a variable of a higher clock are surely executed before the substeps that read this variable, it is true by construction.
Consider the example given in Figure 6, where the first clock block computes the greatest common divisor of inputs a and b. The second clock block sets the output o1 by using gcd and the third block sets o2 by using gcd and o1. The read and write dependencies of the variables of higher clocks are illustrated in Figure 7. The dependencies are acyclic, thus if we schedule all substeps related to C2 after the substeps of C1, the value of gcd is guaranteed to be computed before. Also, if we schedule the substeps of C3 after the substeps of C2, o1 will also be known. In this second case, o1 might be determined or not depending on the context. However, since no other block can set it in our example, we know its value.

We obtain a different result for the example given in Figure 3. Its dependencies are shown in Figure 8. This program has (statically) cyclic dependencies so that any static schedule does not fulfill the requirements imposed by the dependencies of the variables in the program. If we take a closer look to the program, we can see that for scheduling this program correctly, the (dynamic) evaluation of the if statement has to be considered first. Therefore, verifying causality with the full procedure given in Section III succeeds: there is a dynamic schedule for every input.

Nevertheless, we choose the given approximation as a check in our compiler. It can be checked in linear time, since no other block can set it in our example, we know its value.

V. SUMMARY

In general, synchronous languages suffer from well-studied causality problems. Due to many research efforts, suitable heuristics are known to efficiently check the causal behavior of programs. In our previous work, we introduced refined clocks to our imperative synchronous language Quartz to overcome problems like unnecessary over-synchronization of independent concurrent behaviors. However, the introduction of refined clocks imposes new causality problems which a compiler must check for a given program. In this paper, we generalized the notion of causality from single-clocked systems to systems using refined clocks. In addition to a formalization, we defined practical procedures, which conservatively approximate the necessary causality checks to increase the efficiency of compilers.

REFERENCES


