

Performing Causality Analysis by Bounded Model Checking

Klaus Schneider and Jens Brandt

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
<http://es.cs.uni-kl.de>

Abstract

Synchronous systems can immediately react to the inputs of their environment which may lead to so-called causality cycles between actions and their trigger conditions. Systems with causality cycles may not be consistent or may become nondeterministic. For this reason, compilers for synchronous languages usually employ special analyses to guarantee a predictable runtime behavior of the considered programs.

In this paper, we show how causality analysis can be formulated as a model checking problem, so that all of the sophisticated algorithms originally developed for model checking can now also be used for causality analysis. To this end, we model the ‘micro step’ behavior of synchronous programs in terms of a transition relation that can be directly used for symbolic model checking. Moreover, we show that the obtained model checking problems can be even decided by bounded model-checking problems so that modern SAT-solvers can be used to efficiently solve the causality problem.

1. Introduction

Synchronous languages [1, 2, 3, 7, 10] follow the *paradigm of perfect synchrony*: reactions (outputs) of the system can immediately occur with actions (inputs) of the environment. This idealized programming model simplifies the semantics of the languages and leads to a simpler estimation of worst-case execution time (WCET) analysis. Synchronous hardware circuits, as well as Mealy and Moore machines, also follow the paradigm of perfect synchrony. However, since synchronous languages additionally allow programs to read their own outputs, mutual dependencies between actions and their trigger conditions may lead to so-called causality cycles [3]. Such cycles may lead to inconsistencies so that no code can be generated. In

many cases, however, causality cycles can be resolved and deterministic code can be generated. To this end, compilers have to analyze the causality of cyclic dependencies.

Causality cycles in hardware circuits (also called combinational cycles or feedback loops) have already been studied in the early seventies [8, 9, 11]. These cycles (called ‘false paths’ in this setting) also occur in high-level synthesis of circuits by sharing common subexpressions [19]. They are also a major concern in the compilation of synchronous languages [2, 3, 4, 7, 13, 15, 16, 17, 18, 20]. In this application area, causality analysis is usually performed by special algorithms that evaluate the actions and their trigger conditions step-by-step in a fixpoint iteration. During this iteration, the algorithms maintain sets of actions that must be and that can be executed in the current step. If the fixpoint is reached and the must and can sets are equal, the system is causally correct, since actions that can be executed must be executed and vice versa. Usually, implementations of these algorithms perform this computations symbolically by a ternary simulation of the systems [6, 18].

However, existing approaches to causality analysis share some drawbacks: First, they only consider events in the data flow and not memorized values in the data flow. Broadcasting of events is the underlying communication mechanism in Esterel, however events are rather untypical for non-reactive systems. For this reason, Esterel additionally provides valued signals, where the value of a signal is memorized (in contrast to its status). However, ternary simulation of hardware circuits [3, 18] only considers stateful control flow. Some subtle issues, e. h. reincarnation of local variables [13], prevent a straightforward generalization of existing approaches to memorized variables. In particular, one has to additionally deal with write conflicts.

Second, higher data structures such as bitvectors and numbers are only supported by a decomposition to

Booleans. While this approach is theoretically elegant, it has two significant drawbacks: The static analysis of these systems is commonly based on BDDs, which often suffer from the state explosion when examining programs with arithmetic computations. What is more problematic is that the analyzed model at the Boolean level may not correspond with the original program with higher data types: For instance, it may be the case that the analysis at the Boolean level may be able to identify single bits of an integer variable so that the causality may be proved. However, it may be the case that this analysis cannot be lifted to the used higher data types, since that abstraction level is coarser.

In this paper, we therefore propose a different approach to analyze the causality of a synchronous program. In contrast to traditional compilation schemes, we do not compile to a macro step model of the system, but to a refined one that reflects the reactive behavior at the micro step level of execution. The information flow and causal dependencies are explicitly modeled so that they can be analyzed by general-purpose model checkers. Our method also deals both with events and memorized variables so that it does not suffer from the first of the above mentioned problems. Moreover, we treat atomic data types as unique values, so that we also avoid the second problem. Moreover, we show that the causality analysis can be even reduced to a satisfiability problem, since we can easily determine upper bounds for the fixpoint iterations of a given program at compile time.

The rest of the paper is organized as follows: In Section 2, we briefly review the known compilations of synchronous programs to a symbolically represented transition system given at the macro step level of abstraction. In Section 3, we refine this transition relation to micro steps and sketching the resulting verification procedure. Finally, we draw some preliminary conclusions.

2. Macro Step Transition Relation

2.1. The Synchronous Language Quartz

Quartz is a synchronous language that shares its main principles with Esterel. A technical difference between both languages that is important in the remainder of this paper is the manipulation of data values: While Esterel has pure and valued signals as well as variables, there are only variables in Quartz. However, variables in Quartz have a storage type which can be either event or memorized. Memorized variables behave like variables in traditional programming languages: They keep their value as long as another assignment changes

it. In contrast, event variables have a default value unless an assignment is made in the current step. Hence, in the context of digital circuits, memorized variables can be seen as registers and event variables as wires.

Both kinds of variables are modified by actions, where Quartz distinguishes between immediate and delayed actions. Immediate actions ($x = \tau$) immediately evaluate the right-hand side expression τ and immediately assign the obtained value to the variable x at the left hand side. Delayed assignments ($next(x) = \tau$) also immediately evaluate the right-hand side expression τ , but assign the obtained value to x only in the following macro step.

A challenging problem for the compilation of synchronous programs are *schizophrenia problems*: As it is possible that the scope of a local variable can be entered more than once in a macro step, the compiler has to generate different incarnations of local variables at compile time. Hence, after the compilation of a synchronous program, there may be more variables than in the given program. These copies are commonly called reincarnations and must be carefully distinguished [14]. Clearly, we also have to deal with this problem in the causality analysis.

2.2. Guarded Actions as Intermediate Format

We have developed a framework called Averest for the compilation, simulation, and verification of Quartz programs¹. The starting point of the algorithm presented in this paper is therefore the Averest Intermediate Format (AIF), a representation of the synchronous program suitable for many applications. In the following, we only explain the parts of AIF that are relevant for this paper. For further details on the compilation from Quartz to AIF, we refer to [5, 12, 14].

In AIF, the control flow is given by a set of equations, and the data-flow is given by a set of guarded actions. Guarded actions are pairs (γ, \mathcal{C}) consisting of an action² \mathcal{C} as described above and the corresponding precondition γ that triggers the execution of \mathcal{C} . The guarded actions are the essential information that we need to define the data flow of a program. The meaning of a guarded action (γ, \mathcal{C}) is that whenever γ holds, then \mathcal{C} is executed. Executing an immediate assignment $x = \tau$ has the effect that immediately the equation $x = \tau$ holds, while the execution of a delayed assignment means that the value of x at the next point of time is equal to the current value of τ .

¹<http://www.averest.org>

²In this paper, actions are only assignments, which may be immediate or delayed, as explained above.

2.3. Macro Step Transition Relation

For the analysis of the synchronous program with model checkers, it is necessary to translate the programs to equivalent transition systems. In particular, translations to symbolic representations of the transition systems are of interest, where formulas are used to describe the initial states and the transition relation. To describe the translation from AIF to these formulas, we first partition the guarded actions according to their left hand side variables. For example, assume that for a variable x , we have obtained the guarded actions $(\gamma_1, x=\tau_1), \dots, (\gamma_p, x=\tau_p)$ with immediate assignments, and the guarded actions $(\chi_1, \text{next}(x)=\pi_1), \dots, (\chi_q, \text{next}(x)=\pi_q)$ with delayed assignments. Then, the following formulas must hold at every point of time³:

$$\left(\bigwedge_{i=1}^p (\gamma_i \rightarrow (x = \tau_i)) \right) \wedge \left(\bigwedge_{i=1}^q (\chi_i \rightarrow (\text{next}(x) = \pi_i)) \right)$$

However, the above formula for the data flow of the variable x is not yet complete. It only determines the value of x at a point of time t when an immediate assignment is executed at time t or a delayed assignment has been executed at time $t - 1$. If neither is the case, then we must also determine a value for x which is done by the reaction-to-absence. In this case, x will have a default value $\text{Default}(x)$ which is defined differently for variables of event and memorized storage classes: For a memorized variable, $\text{Default}(x)$ is the previous value of x , and for an event variable, a value determined by the type of x is chosen (which is 0 for numbers and `false` for booleans). A subtle problem is imposed by reincarnated variables: We have to transfer the value of the most recent incarnation to the actual local variable, which is not explicitly encoded in the guarded actions (see [12, 14] for more details).

In the following, we do not consider output variables. Instead, we only consider the more general⁴ case where x is a local variable with some reincarnations. For each variable, there are immediate actions, delayed actions and immediate actions on the reincarnations of the variable, but no delayed actions on the reincarnations (since reincarnations happen immediately). To describe this formally, assume we have computed for a local variable x and its d reincarnations x_1 (innermost), \dots, x_d (outermost) the corresponding guarded actions as shown in the upper part of Figure 1.

As there are no delayed actions on reincarnated variables, the transition relation of a reincarnated vari-

³Note that this formula can become false if there are write conflicts, which can be verified with a model checker.

⁴The case of global output variables is a special case of local variables where no reincarnations have to be considered.

able x_i is defined as the invariant Invar_{x_i} in Figure 1 that has to hold at every point of time. This invariant simply states that whenever the trigger condition $\gamma_{i,j}$ holds, then the corresponding equation $x_i = \tau_{i,j}$ must also hold. If no trigger condition $\gamma_{i,j}$ holds, then it is required that x_i equals to the default value $\text{Default}(x)$.

The *initial condition* Init_x of the variable x is constructed analogously to Invar_{x_i} which indicates that the semantics of reincarnated variables corresponds with an initialization of a variable. The explanations for the construction of Init_x are the same as for Invar_{x_i} .

The *transition relation* of x is more difficult: We have to distinguish between event and memorized variables x due to the different reaction to absence. In any case, the equation $x = \tau_j$ must hold if the trigger condition γ_j holds. Moreover, if the trigger condition χ_j of a delayed assignment $\text{next}(x) = \pi_j$ holds, then x must have the value π_j at the next point of time (note that π_j is evaluated with the current variables to determine the value of x for the next point of time).

If neither a trigger condition γ_j of an immediate assignment nor a trigger condition of a delayed assignment χ_j holds, then we have to implement the *reaction-to-absence*: The value of an event variable x is the default value $\text{Default}(x)$ associated with the type of x .

The reaction-to-absence for memorized variables is more difficult. Clearly, if an immediate or delayed assignment is executed, we have the same effect as for event variables, and therefore the first two conjuncts of Trans_x are the same for event and memorized variables. The reaction-to-absence is however more difficult for memorized variables, since there may be an interference of reincarnated variables and the original memorized variable x : It may be the case that there is neither a delayed assignment to x in the surface of the local declaration nor an immediate assignment right after this point of time. In this case, x has to capture the value of that reincarnated variable x_i that corresponds with the surface that has been executed most recently.

Hence, we have to determine the ‘most recently executed surface’ in order to capture the value of the most recently reincarnated variable. This is the reason why the preconditions of each surface that has been generated by a loop has been stored in the intermediate file during the compilation. Using these preconditions, we are able to check one after the other (in the ordering of their nestings) to determine the most recently activated surface. Note that this selection is done dynamically and the formulas we generate therefore have to cover all possible cases.

Hence, assume that g_{o_i} is the precondition of a loop

surface where the local variable x has been reincarnated to x_i (hence, $\bigvee_{j=1}^{p_i} \gamma_{i,j} \rightarrow go_i$ must be valid, but the converse may not hold). The transition relation for x is then determined by the expression $\text{Initialize}(x)$. With the case construct, we check the preconditions go_i one after the other. Note that if go_i is the first one that holds, then all go_j with $j < i$ do also hold, which means that go_i refers to the outermost surface whose reincarnated variable has to be referred to. Finally, if no precondition go_i holds, then the control flow moved from somewhere inside the scope of x without leaving the scope in between. Hence, we simply store the previous value of x , which is the default case of the case construct in the expression $\text{Initialize}(x)$.

This concludes the construction of the transition relation for a single variable. The whole data flow is simply a conjunction of all output and local variables of the program together with the equations of the control flow.

3. Micro Step Transition Relation

In the previous section, we showed how the transition relation for the data flow is determined at the abstraction level of macro steps. However, while the abstraction level of macro steps directly corresponds with the synchronous semantics, it is too coarse to verify that the abstraction from the asynchronous micro step to the synchronous macro step level is correct. For this reason, we have to refine the transition relation to a micro step level that incorporates the SOS reaction rules given in [3, 12] that determine the constructive reaction of a program.

At this level, we explicitly model the progress of information since the values of the modifiable variables are determined step by step in terms of micro steps. To this end, we formally have to add an explicit unknown value \perp to every data type, so that we can express that the value of a variable is not yet known at a particular micro step.

3.1. Modeling the Progress of Information

Due to the introduction of \perp for every data type, we have to represent the unknown value \perp also in the transition relation. Note that a variable has this value if its actual value in this macro step has not yet been determined. For this reason, we embed the unknown value \perp explicitly as follows: For every local, reincarnated local, and output variable x , we add a boolean-typed variable $\text{kvar}(x)$ that holds iff the value of x is known, i.e. if x has not value \perp . Hence, the pair $(\text{kvar}(x), v)$ encodes the actual value of the variable x :

$x \wedge \text{false} = \text{false}$	$\text{false} \wedge x = \text{false}$
$x \vee \text{true} = \text{true}$	$\text{true} \vee x = \text{true}$
$\text{false} \rightarrow x = \text{true}$	$x \rightarrow \text{true} = \text{true}$
$(x \Rightarrow y y) = y$	
$(\text{true} \Rightarrow x y) = x$	$(\text{false} \Rightarrow x y) = y$
$x \cdot 0 = 0$	$0 \cdot x = 0$
$0 \leq \tau_{\mathbb{N}} = \text{true}$	$\tau_{\mathbb{N}} < 0 = \text{false}$

Figure 2. Lazy Evaluation Rules

if $\text{kvar}(x)$ holds, then the value of x is known and is v , otherwise the value of x is not yet known and we have to ignore the content of v .

Using the variables $\text{kvar}(x)$, we can explicitly model the progress of the information flow, which is obtained by evaluating the program expression step by step until either assignments can be executed that determine the current value of a variable or until it becomes clear that no assignment will modify the current value of a variable so that the reaction to absence will determine it. This progress of the information flow also makes use of lazy evaluation (Some examples are shown in Figure 2.), which has to be encoded in the transition relation at the micro step level.

To this end, we formally define a function that maps a program expression σ of arbitrary type to a boolean formula $k(\sigma)$ such that $k(\sigma)$ holds iff the expression σ can be evaluated to a known value. Formally, the formula $k(\sigma)$ follows the rules of Figure 2 and is defined as shown below:

- for variables and constants, we define
 - $k(x) := \begin{cases} \text{true} & \text{: if } x \text{ is an input variable} \\ \text{kvar}(x) & \text{: otherwise} \end{cases}$
 - $k(c) := \text{true}$
- for boolean operators, we define:
 - $k(!\varphi) := k(\varphi)$
 - $k(\varphi \ \& \ \psi) := \begin{pmatrix} k(\varphi) \wedge k(\psi) \vee \\ k(\varphi) \wedge (\varphi = \text{true}) \vee \\ k(\psi) \wedge (\psi = \text{true}) \end{pmatrix}$
 - $k(\varphi \ | \ \psi) := \begin{pmatrix} k(\varphi) \wedge k(\psi) \vee \\ k(\varphi) \wedge (\varphi = \text{false}) \vee \\ k(\psi) \wedge (\psi = \text{false}) \end{pmatrix}$
- for arithmetic operators, we define:
 - $k(-\tau) := k(\tau)$
 - $k(\text{abs}(\tau)) := k(\tau)$
 - $k(\tau + \pi) := k(\tau) \wedge k(\pi)$
 - $k(\tau - \pi) := k(\tau) \wedge k(\pi)$
 - $k(\tau * \pi) := \begin{pmatrix} k(\tau) \wedge k(\pi) \vee \\ k(\tau) \wedge \tau = 0 \vee \\ k(\pi) \wedge \pi = 0 \end{pmatrix}$

- $k(\tau / \pi) := k(\tau) \wedge k(\pi)$
- $k(\tau \% \pi) := k(\tau) \wedge k(\pi)$

- for arithmetic relations, we define

- $k(\tau \leq \pi)$
 $:= \begin{cases} k(\tau) \wedge k(\pi) & \text{: for signed types} \\ k(\tau) \wedge (\tau = 0 \vee k(\pi)) & \text{: for unsigned types} \end{cases}$
- $k(\tau < \pi) := k(!(\pi \leq \tau))$

- for equality and inequality operators, we define

- $k(\tau == \pi) := k(\tau) \wedge k(\pi)$
- $k(\tau != \pi) := k(\tau) \wedge k(\pi)$

Clearly, in the behavior of the program, we can only make use of an expression if we know its value. For this reason, the entire execution of the actions is controlled by the data flow: We start with unknown values for output, local reincarnated, and local variables and try to determine their values with the micro steps that can occur in a macro step. It is known that the program is constructive if and only if there is a dynamic schedule to execute the micro steps in such a way that all values become known step by step. The procedure is repeated when we enter a new macro step.

For this reason, we introduce a clock signal tick, which is true whenever all variables have become known values. If this happens, the delayed assignments are executed, the location variables change their values according to the control flow, and the input variables are allowed to change their values in a nondeterministic way (which reflects the uncontrollable input of the environment).

Macro steps are therefore separated by occurrences of the tick signal, and between two clock ticks, the micro steps of a macro step are executed: As long as tick is false, the immediate assignments to the variables are executed if the values of the guards and right hand side expressions are known, and the guard is true.

We therefore distinguish between the *information flow* and the *data flow*. The information flow of a variable x is determined by the corresponding variable $k(x) = \text{kvar}(x)$ that holds if and only if the value of x is already determined in the current macro step.

The transition relation of the information flow can be formulated as an equation system as shown in Figure 3 that contains the following cases:

- If there is no clock tick, the value of x remains known if it was already known⁵.

⁵Although a guarded action $(\gamma_j, x=\tau_j)$ that once fires in a macro step remains enabled until the end of the macro step, we have to add this constraint, since the variable x may be known due to a delayed assignment of the previous macro step.

- If there is no clock tick, the value of x becomes known if a guarded action $(\gamma_j, x=\tau_j)$ with an immediate assignment $x=\tau_j$ can be fired. This is the case if and only if the value of the guard γ_j is known to be true and if the value of the right hand side expression τ_j is known.
- If there is no clock tick, and all guards γ_j are known to be false, the reaction to absence determines the value of x : The formula of Figure 3 simply demands that $\text{next}(x)=x$ has to hold in this case, since x has been given the now desired value at the previous clock tick as a preliminary value.
- If there is a clock tick, then the values of all variables are known. Therefore, we can execute all enabled delayed actions. If one of the delayed actions can be fired, then the value of x is known for the following macro step.
- Otherwise, the value of x is not known.

The data flow of x is determined by the same cases as formalized in formula ValTrans_x given in Figure 3:

- If there is no clock tick and a guarded action $(\gamma_j, x=\tau_j)$ with an immediate assignment $x=\tau_j$ can be fired, then x will receive the value of τ_j at the next point of time. Note that if more than one guarded action can be fired with different values τ_i and τ_j , then there is no transition.
- If there is no clock tick and no immediate guarded action can be fired, then x will keep its value. This covers several cases: First, if the reaction-to-absence should take place, since all guards γ_j are known to be false, then keeping the value of x is correct, since we already provided the desired value for x at the previous clock tick: Either there was a delayed assignment in the previous macro step that determined the current value of x , or x has been initialized according to $\text{Initialize}(x)$.
- If there is a clock tick, then the values of all variables are known, and therefore, we can execute all enabled delayed actions. If one of the delayed actions can be fired, then the value of x is known for the following macro step. Note again that there is no transition if several delayed guarded actions with different values π_i and π_j are fired.
- Finally, if there is a clock tick, but no delayed action can be fired, then x is initialized according to $\text{Initialize}(x)$.

Note that reincarnated variables always have event storage mode and no delayed actions, which simplifies their transition relation accordingly. Finally, note that the case distinctions as given by the left hand sides of the implications of ValTrans_x and $\text{ReincarTrans}_{x_i}$ are

Propagation of Knowledge During Clock Cycle:

$$\text{KnownTrans}_{\mathbf{x}_i} := \left(\text{next}(k(\mathbf{x}_i)) : \Leftrightarrow \neg\text{tick} \wedge \left(\begin{array}{l} k(\mathbf{x}_i) \vee \\ \left(\bigvee_{j=1}^{p_i} k(\gamma_{i,j}) \wedge \gamma_{i,j} \wedge k(\tau_{i,j}) \right) \vee \\ \left(\bigwedge_{j=1}^{p_i} k(\gamma_{i,j}) \wedge \neg\gamma_{i,j} \right) \end{array} \right) \right)$$

$$\text{KnownTrans}_{\mathbf{x}} := \left(\text{next}(k(\mathbf{x})) : \Leftrightarrow \left(\begin{array}{l} \neg\text{tick} \wedge k(\mathbf{x}) \vee \\ \neg\text{tick} \wedge \left(\bigvee_{j=1}^p k(\gamma_j) \wedge \gamma_j \wedge k(\tau_j) \right) \vee \\ \neg\text{tick} \wedge \left(\bigwedge_{j=1}^p k(\gamma_j) \wedge \neg\gamma_j \right) \vee \\ \text{tick} \wedge \bigvee_{j=1}^q \chi_j \end{array} \right) \right)$$

Micro Step Transition Relation of Reincarnated Variable \mathbf{x}_j :

$$\text{ReincarTrans}_{\mathbf{x}_i} := \left(\left(\begin{array}{l} \neg\text{tick} \rightarrow \left(\bigwedge_{j=1}^{p_i} (k(\gamma_{i,j}) \wedge \gamma_{i,j} \wedge k(\tau_{i,j}) \rightarrow \text{next}(\mathbf{x}_j) = \tau_{i,j}) \right) \\ \neg\text{tick} \rightarrow \left(\neg \left(\bigvee_{j=1}^{p_i} k(\gamma_{i,j}) \wedge \gamma_{i,j} \wedge k(\tau_{i,j}) \right) \rightarrow \text{next}(\mathbf{x}_j) = \text{Default}(\mathbf{x}) \right) \\ \text{tick} \rightarrow \text{next}(\mathbf{x}_j) = \text{Default}(\mathbf{x}) \end{array} \right) \right) \wedge$$

Micro Step Transition Relation of Local/Output Variable \mathbf{x} :

$$\text{ValTrans}_{\mathbf{x}} := \left(\left(\begin{array}{l} \neg\text{tick} \rightarrow \left(\bigwedge_{j=1}^p (k(\gamma_j) \wedge \gamma_j \wedge k(\tau_j) \rightarrow \text{next}(\mathbf{x}) = \tau_j) \right) \\ \neg\text{tick} \rightarrow \left(\neg \left(\bigvee_{j=1}^p k(\gamma_j) \wedge \gamma_j \wedge k(\tau_j) \right) \rightarrow \text{next}(\mathbf{x}) = \mathbf{x} \right) \\ \text{tick} \rightarrow \left(\bigwedge_{j=1}^q (\chi_j \rightarrow \text{next}(\mathbf{x}) = \pi_j) \right) \\ \text{tick} \rightarrow \left(\neg \bigvee_{j=1}^q \chi_j \rightarrow \text{next}(\mathbf{x}) = \text{Initialize}(\mathbf{x}) \right) \end{array} \right) \right) \wedge$$

where $\text{Initialize}(\mathbf{x})$ is defined as in Figure 1 on Page 4

Figure 3. Micro Step Transition Relation to Define the Data Flow of a Local/Output Variable \mathbf{x}

complete; that is the disjunction of the left hand sides of the implications is valid.

The formulas given in Figure 3 describe the micro step transition relation of a particular local/output variable x together with its potential reincarnated variables. In addition to this, we also have to determine the flow of the input and location variables, and we have to define the clock signal. The latter is defined as follows:

$$\text{ClockTrans} ::= \left(\text{tick} : \Leftrightarrow \bigwedge_{\mathbf{x}} k(\mathbf{x}) \wedge \bigwedge_{j=1}^d k(x_j) \right)$$

The new clock tick can arrive as soon as all values of the local and output variables become known. We then perform the following actions to initiate the next macro step:

- The *values of the location variables* ℓ can be computed in terms of the now known value of the inputs, outputs, and local variables. Hence, we add for every location variable ℓ with transition equation $\text{next}(\ell) = \varphi_\ell$ the following transition equation $\text{next}(\ell) = (\text{tick} \Rightarrow \varphi_\ell | \ell)$.
- The *input variables* are read from the environment, which is done by allowing the input variables to change their values in a nondeterministic way. However, when tick is false, we have to make sure that the inputs do not change any more. Therefore, we have to add the formula $\neg \text{tick} \rightarrow \text{next}(x) = x$ for every input variable x .

It is easily seen that the micro step behavior as formalized in Figure 3 and in the above additional explanations for the clock definition, the location variables and the input variables describe the semantics of an *asynchronous circuit*. Moreover, the entire behavior is described in terms of *conditional rewrite rules*, so that also term rewriting techniques are adequate for the analysis of the micro step behavior of Quartz programs.

3.2. Relation to Classic Causality Analysis

It is interesting to consider the classic case, i.e., the case where x is an event variable that only has immediate assignments with the right hand side `true`. In this case, the formulas of Figure 3 reduce to the ones shown in Figure 4.

As can be seen in Figure 4, the transition relation can now be given in form of an equation system. The reason for this is that at every clock tick, the known values as well as the values themselves are reset to false,

and during the clock cycle, the value of such an event signal becomes true if and only if one of the guards γ_j is known to be true. The known value becomes additionally true if all guards γ_j are known to be false. As a consequence, x implies $k(x)$.

The equational form given in Figure 4 is interesting because it gives a new insight in the causality analysis in terms of a fixpoint computation: The reason why we can obtain an equational form is that there are by definition no write conflicts if we only have assignments of the form $x = \text{true}$, which is the case in classic Esterel. In principle, we can therefore also allow delayed assignments of the form $\text{next}(x) = \text{true}$, and we could still derive equation systems as it has been shown in [13]. However, this requires the introduction of new state-holding variables since we have to store if a delayed assignment was enabled in the previous macro step.

3.3. Causality Analysis as Model Checking

In the previous sections, we showed how the known transition relation at the macro step level can be refined to the level of micro steps. In this section, we sketch the resulting verification tasks for causality analysis and show how these can be even reduced to a simple bounded model checking problem.

Obviously, a program is causally correct, if in each macro step, the values of all variables can be determined for all possible inputs. Due to the definition of `ClockTrans` (see Section 3.1), the states in which all variables are known can be identified by the tick variable, which also marks the beginning of a new macro step. Hence, a single transition at the macro step model is replaced by a finite chain of transitions in the micro step model leading from one state where tick holds to another state where tick holds. Causally incorrect transitions (which do not exist in the macro step model), are chains that do not lead to a valid new state, i.e. tick does not appear from a certain point on. Such execution sequences end in a self-loop of a state without a tick or these executions are finite. Thus, any system that is guaranteed to hit a tick state infinitely often, is causally correct. This property can be simply denoted by GF tick in linear time temporal logic and can be checked by any state-of-the-art model checker that supports linear time logic (LTL).

Furthermore, it can be observed that in the micro step transition relation shown in Figure 3, at least one variable becomes known in each micro step (or a fixpoint is already reached). Thus, there is always a progress of information for causally correct systems. Hence, the number of transitions between two tick signals is bounded by the number of output, reincarnated

- [4] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis Cedex (France), September 1998.
- [5] J. Brandt. The Averest intermediate format v2. Internal report, Department of Computer Science, University of Kaiserslautern, 2008.
- [6] R. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In E. Clarke and R. Kurshan, editors, *Computer Aided Verification (CAV)*, volume 531 of *LNCS*, pages 33–43, New Brunswick, NJ, USA, 1991. Springer.
- [7] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [8] D. Huffman. Combinational circuits with feedback. In A. Mukhopadhyay, editor, *Recent Developments in Switching Theory*, pages 27–55. Academic Press, 1971.
- [9] W. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Transactions on Computers*, C-19(2):162–166, February 1970.
- [10] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *IEEE*, 79(9):1321–1336, 1991.
- [11] R. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers*, C-26(6):606–607, 1977.
- [12] K. Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, 2008.
- [13] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, DC, USA, 2004. ACM.
- [14] 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.
- [15] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, United Kingdom, 2005.
- [16] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
- [17] E. Sentovich. Quick conservative causality analysis. In *International Symposium on System Synthesis (ISSS)*, pages 2–8, Antwerp, Belgium, 1997. IEEE Computer Society.
- [18] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC)*, Paris, France, 1996. IEEE Computer Society.
- [19] L. Stok. False loops through resource sharing. In *Conference on Computer Aided Design (ICCAD)*, pages 345–348. IEEE Computer Society, 1992.
- [20] Y. Zhou and E. Lee. A causality interface for deadlock analysis in dataflow. In S. Min and W. Yi, editors, *International Conference on Embedded Software (EMSOFT)*, pages 44–52, Seoul, Korea, 2006. ACM.