Formal Reasoning About Causality Analysis

Jens Brandt and Klaus Schneider

University of Kaiserslautern
Embedded Systems Group, Department of Computer Science
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract. Systems that can immediately react to their inputs may suffer from cyclic dependencies between their actions and the corresponding trigger conditions. For this reason, causality analysis has to be employed to check the constructiveness of the programs which implies the existence of unique and consistent behaviours. In this paper, we describe the embedding of various views of causality analysis into the HOL4 theorem prover to check their equivalence. In particular, we show the equivalence between the classical analysis procedure, which is based on a fixpoint computation, and a formulation as a (bounded) model checking problem.

1 Introduction

For the modelling of embedded systems, or more generally, the modelling of systems with concurrent actions whose execution may consume time, many *models of computation* [8,10,12] have been considered. Among these models of computation are asynchronous models like dataflow process networks and Hoare's CSP, discrete-event models as used by most hardware description languages including VHDL, Verilog and SystemC, and synchronous models as used by synchronous hardware circuits, synchronous programming languages [3] and classical automata theory.

Most of these models of computation define a *causality* relation between actions and their trigger conditions. Roughly speaking, causality determines a logical sequentiality between trigger conditions and subsequent reactions according to the stimuli of the environment. Causality can be achieved in various ways: for example, the notion of δ-time has been introduced in hardware description languages to circumvent causality problems, and tagged tokens have been introduced in dataflow computers to solve this problem.

In synchronous models of computation, however, causality is not given by construction. There is neither a finer notion of time as given by δ-time nor are there tagged values to establish a causality relation. Instead, the execution of a synchronous system is partitioned into macro steps that consist of finitely many micro steps. The values of the variables remain constant during all micro steps within a macro step and change synchronously when proceeding to the next macro step. Time is given as a logical time in the number of macro steps. Hence, causality analysis is much more difficult than in other models of computation.

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 118–133, 2008.
© Springer-Verlag Berlin Heidelberg 2008

For this reason, special procedures for causality analysis have been developed for synchronous languages [4,22]. It is well-known that a synchronous program is causally correct (constructive) if and only if for all inputs, there is a dynamic schedule of the micro step actions in all macro steps (hence, the program can be executed on a sequential machine). Moreover, it is known that causality analysis is equivalent to the stability analysis of combinational feedback loops in hardware circuits [11,9,17,13,22,4,15,16,14] under Brzozowski and Seger's unbounded delay model [5,22].

Since the causality of a program depends on its syntax (or equivalently on the particular structure of a hardware circuit) and not only on its semantics, the causality analysis furthermore depends on the translation to semantically equivalent hardware circuits or sequential programs. We have studied such variants in depth in our previous research [19,20,21]. As it turned out that the problem is very subtle and depends on details of the used definitions, we believe that the research in this area would greatly benefit from a solid formal treatment using an interactive theorem prover like HOL4. Hence, this paper is the first step towards such a formal treatment, and it already presents an equivalence proof between two kinds of causality analyses, namely one that could be used at run-time with concrete inputs and a static one, which is used at compile-time with symbolic inputs (so that a symbolic analysis is obtained). The symbolic causality analysis is the second added value of this paper, since symbolic methods only exist for the special case of Boolean event variables so far.

In this paper, we do not consider a particular language like our synchronous programming language Quartz [18]. Instead, we consider a language-independent formalisation of synchronous systems that is based on so-called guarded actions to obtain a theory that is as general as possible. A guarded action is thereby a pair $(\gamma, \mathcal{C})$ where the guard γ is a Boolean condition, and where the action $\mathcal{C}$ is an atomic action of the system under consideration. Throughout this paper, we only consider immediate assignments of the form $y = \tau$, where y is an output variable, and τ is an expression that is type-consistent with y. Further kinds of actions like assumptions, assertions, and delayed assignments [18] do not further complicate the causality analysis [19]. Moreover, we do not take into account the reachability of states, which is necessary in a full causality analysis by a compiler. Again, this would not further complicate the algorithms. Finally, we only consider variables of type Boolean, while a real programming language offers typically further types. However, types do not complicate our symbolic analysis either.

Guarded actions as the ones considered here have been widely used in computer science so far, and entire programming languages like Unity [6,1,2] have been built on top of them. However, the semantics of the guarded actions often differs, depending on the preferred model of computation. As we consider the synchronous model of computation, the system's computation consists of micro and macro steps: Thus, we must analyse the causal order of the micro steps within one macro step, and variables have constant values during the analysis. Indeed, the major goal of causality analysis is to find a constructive schedule (dependent on the values of the input variables) to determine the successive values of the output variables.

The symbolic causality analysis of this synchronous system starts with known inputs x_i and unknown outputs y_i. For this reason, the initial environment $\mathcal{E}_0$ maps all input variables to known Boolean constants, while it maps the output variables to a third

value $\bot$ that means 'still unknown' in the multi-valued causality analysis. Starting with the initial environment $\mathcal{E}_0$, the following iteration is performed in causality analysis to compute an environment $\mathcal{E}_{i+1}$ from a given environment $\mathcal{E}_i$. We first evaluate all guards of the actions and partition the actions into (1) must-actions that have a guard $\mathcal{E}_i(\gamma) = \mathsf{true}$ and (2) can-actions that have a guard $\mathcal{E}_i(\gamma) \in \{\mathsf{true}, \bot\}$, respectively. Then, we 'execute' the must-actions, so that the assigned variables' values are updated in the new environment $\mathcal{E}_{i+1}$. Moreover, if all actions of an output variable y are cannot-actions, we can assign a default value to y (which may be a fixed constant in case of event variables (transient) or the previous value in case of memorised variables (persistent)). In all other cases, we have to maintain the current value of y, which may still be unknown $\bot$. If a variable y should already have a known value, but a must-action assigns a different value, we update the value of y to $\top$ to indicate a write conflict.

$$
\left\{
\begin{array}{l}
(x_0 \wedge y_5, y_0 = \mathsf{true}),\\
(x_1 \vee y_0, y_1 = \mathsf{true}),\\
(x_2 \wedge y_1, y_2 = \mathsf{true}),\\
(x_0 \vee y_2, y_3 = \mathsf{true}),\\
(x_1 \wedge y_3, y_4 = \mathsf{true}),\\
(x_2 \vee y_4, y_5 = \mathsf{true})
\end{array}
\right\}
\qquad
\left\{
\begin{array}{l}
y_0 = x_0 \wedge y_5\\
y_1 = x_1 \vee y_0\\
y_2 = x_2 \wedge y_1\\
y_3 = x_0 \vee y_2\\
y_4 = x_1 \wedge y_3\\
y_5 = x_2 \vee y_4
\end{array}
\right.
\qquad
\left\{
\begin{array}{l}
y_0 = x_0 \wedge (x_2 \vee x_1)\\
y_1 = x_1 \vee (x_0 \wedge x_2)\\
y_2 = x_2 \wedge (x_1 \vee x_0)\\
y_3 = x_0 \vee (x_2 \wedge x_1)\\
y_4 = x_1 \wedge (x_0 \vee x_2)\\
y_5 = x_2 \vee (x_1 \wedge x_0)
\end{array}
\right.
$$

Fig. 1. An Example of a Cyclic Equation System due to [17]

In case of Boolean event variables, a set of guarded actions is equivalent to a (potentially cyclic) equation system, i.e., a combinational hardware circuit with potential feedback loops. As an example (due to Rivest [17]), consider the guarded actions given on the left hand side of Figure 1 for inputs x_0, x_1 and x_2 and outputs y_0, y_1, y_2, y_3, y_4 and y_5. This set of guarded actions is equivalent to the cyclic Boolean equation system shown in the middle of Figure 1. Using ternary simulation [13,22,4,19], it is possible to convert every causally correct set of guarded actions into an acyclic equation system. In case of our example of Figure 1, this equivalent acyclic equation system is given on the right of Figure 1.

While there are paper-and-pencil proofs for the equivalence of the causality analysis based on can-must analysis and the symbolic ternary simulation, there is no publication on a symbolic analysis for non-Boolean or non-event variables. In this paper, we present such a symbolic analysis as a generalisation of ternary simulation. To this end, we have to take into account the problem of write conflicts which does not appear for Boolean events (since disjunction is used as an implicit conflict resolution function). In addition, we prove the equivalence between the can-must causality analysis and our symbolic formulation based on extending each data type by the constants $\bot$ and $\top$.

To summarise, we present in this paper the formalisation of the traditional causality analysis for synchronous systems as well as the definition of a fully symbolic version of a general causality analysis. We use the HOL4 theorem prover to prove the equivalence of these two variants of the causality analysis, so that we can guarantee that what is checked by means of model checking in a compiler with the symbolic analysis exactly matches the definition of causality analysis as given by the can-must analysis.

Moreover, we extend the classical analysis by considering run-time errors like write conflicts, division by zero, or access to array elements outside the declared range. This is accomplished by generalising the traditional three-valued analysis to a four-valued setting using a constant $\top$ for 'run-time error' in addition to $\bot$ 'yet unknown'.

This paper is organised as follows: In Section 2 we present a formalisation of the traditional can-must analysis. Section 3 describes our symbolic approach to causality analysis, which is subsequently formalised and shown to be equivalent in Section 4. Finally, Section 5 draws some conclusions.

2 Formalisation of Traditional Can-Must Analysis

Before we formalise the traditional can-must analysis in the last part of this section, we first have to model the system description, which is based on guarded actions (Section 2.1) and the environment to allow the evaluation of terms (Section 2.2).

2.1 System Description

The main objective of our work is the reasoning about different procedures for causality analysis; we are less interested in causality analysis of a particular system. Therefore, we need a *deep embedding* of the system description to quantify over systems in the logic. Hence, our first step is the formalisation of the guarded actions.

A guarded action is a pair $(\gamma, x = \tau)$, where $\mathsf{grd}(\gamma, x = \tau) = \gamma$ is called the guard, $\mathsf{lhs}(\gamma, x = \tau) = x$ the left-hand side, and $\mathsf{rhs}(\gamma, x = \tau) = \tau$ the right-hand side of the action. The guard and the right-hand side are Boolean expressions, which are defined in HOL as follows:

$$\begin{aligned}\texttt{BlExpr} \vdash_{\text{def}} & \mathit{false} \in \mathit{BlExpr} \mid \mathit{true} \in \mathit{BlExpr} \mid x, x \in \mathcal{V} \\ & \mid \mathit{not}(e), e \in \mathit{BlExpr} \\ & \mid \mathit{and}(e_1, e_2), e_1, e_2 \in \mathit{BlExpr} \mid \mathit{or}(e_1, e_2) e_1, e_2 \in \mathit{BlExpr}\end{aligned}$$

2.2 Four-Valued Environment

The previous subsection only describes the syntax of synchronous systems in our HOL theory. To define their semantics, we have to formalise the environment of the system. In the traditional can-must analysis, the environment maps each variable to a three-valued truth value. We extend this definition here to integrate write conflicts and other run-time errors into the causality analysis. Thus, the environment maps each variable to one of the four truth values $\mathbb{F} = \{\bot, 0, 1, \top\}$.

	$\ddot{\neg}$
$\bot$	$\bot$
0	1
1	0
$\top$	$\top$

$\ddot{\wedge}$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	$\bot$	$\top$
0	0	0	0	$\top$
1	$\bot$	0	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\ddot{\vee}$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\mathsf{sup}_{\mathbb{F}}$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Fig. 2. Definitions of Four-Valued Negation, Conjunction and Disjunction

In the rest of this section we implement a theory based on four-valued logic. To reason about monotonic functions, and the existence of fixpoints, we define a strict partial order $\ddot{<}$ on the type $\mathbb{F}$ as follows: $x \ddot{<} y :\Leftrightarrow x \neq y \wedge (x = \bot \vee y = \top)$. It is easily seen that $(\mathbb{F}, \ddot{<})$ is a complete lattice, since two elements have a supremum and an infimum (see Figure 2).

Clearly, our theory contains four-valued generalisations for all Boolean operators that are defined according to the truth tables shown in Figure 2. As can be seen, all four-valued operations are maximal monotonic generalisations of the corresponding Boolean operations (with respect to our partial order). The reason for this choice will be discussed later in this paper, when we show the equivalence of this method to our model-checking approach.

The environment itself is defined as a finite map [7] from natural numbers, which represent variables in our case, to four-valued truth values. The actual implementation is hidden behind the functions $\mathcal{E}(x)$ and $\mathcal{E}_x^\tau$, which read and update the value of a given variable x in the environment $\mathcal{E}$, respectively. The evaluation $[\![\tau]\!]_\mathcal{E}^\mathbb{F}$ of an expression τ to a value in $\mathbb{F}$ with respect to the given environment $\mathcal{E}$ is defined recursively as follows:

$$
\begin{aligned}
&\texttt{blEval4\_def} \vdash_{\text{def}} \\
&\quad ([\![\mathit{false}]\!]_\mathcal{E}^\mathbb{F} = 0) \wedge ([\![\mathit{true}]\!]_\mathcal{E}^\mathbb{F} = 1) \wedge \\
&\quad ([\![x]\!]_\mathcal{E}^\mathbb{F} = \mathcal{E}(x)) \wedge ([\![\mathit{not}(\tau)]\!]_\mathcal{E}^\mathbb{F} = \ddot{\neg}[\![\tau]\!]_\mathcal{E}^\mathbb{F}) \wedge \\
&\quad ([\![\mathit{and}(\tau_1, \tau_2)]\!]_\mathcal{E}^\mathbb{F} = [\![\tau_1]\!]_\mathcal{E}^\mathbb{F} \ddot{\wedge} [\![\tau_2]\!]_\mathcal{E}^\mathbb{F}) \wedge ([\![\mathit{or}(\tau_1, \tau_2)]\!]_\mathcal{E}^\mathbb{F} = [\![\tau_1]\!]_\mathcal{E}^\mathbb{F} \ddot{\vee} [\![\tau_2]\!]_\mathcal{E}^\mathbb{F})
\end{aligned}
$$

2.3 Fixpoint Iteration

As already explained in the introduction, the can-must analysis iteratively computes values for all output variables of the environment $\mathcal{E}$ until a fixpoint is reached. To this end, we have to maintain two sets of guarded actions $\mathcal{A}$: The guard of the *must-actions* is true, while the guard of the *can-actions* is not false.[1] To this end, we provide the following definitions:

$$
\begin{aligned}
&\texttt{can4\_def} \vdash_{\text{def}} \\
&\quad (\mathsf{can}(\bot) = \mathsf{true}) \wedge (\mathsf{can}(0) = \mathsf{false}) \wedge \\
&\quad (\mathsf{can}(1) = \mathsf{true}) \wedge (\mathsf{can}(\top) = \mathsf{false}) \\
&\texttt{canActs\_def} \vdash_{\text{def}} \\
&\quad \mathsf{canActs}_\mathcal{E}(\mathcal{A}) = \textbf{filter}\ \lambda(\gamma, x = \tau).\ \mathsf{can}([\![\gamma]\!]_\mathcal{E}^\mathbb{F})\ \textbf{from}\ \mathcal{A} \\
&\texttt{must4\_def} \vdash_{\text{def}} \\
&\quad (\mathsf{must}(\bot) = \mathsf{false}) \wedge (\mathsf{must}(0) = \mathsf{false}) \wedge \\
&\quad (\mathsf{must}(1) = \mathsf{true}) \wedge (\mathsf{must}(\top) = \mathsf{true}) \\
&\texttt{mustActs\_def} \vdash_{\text{def}} \\
&\quad \mathsf{mustActs}_\mathcal{E}(\mathcal{A}) = \textbf{filter}\ \lambda(\gamma, x = \tau).\ \mathsf{must}([\![\gamma]\!]_\mathcal{E}^\mathbb{F})\ \textbf{from}\ \mathcal{A}
\end{aligned}
$$

After the set of guarded actions is partitioned into must-, cannot-, and the remaining actions, their update of the environment is defined as follows: for must-actions, the variable on the left-hand side should be assigned the supremum of its old value and the right-hand side. This construction ensures two important properties. First, the order of

[1] For a more intuitive formalisation, we use *can-actions* and not the complement *cannot-actions*, which can be usually found in traditional causality.

assignments to the same variable is irrelevant, which is a result of the associativity and commutativity of the $\mathsf{sup}_{\mathbb{F}}$ operation. Thus, multiple actions can be executed sequentially without caring about the sequential order. Second, instead of changing a variable from one Boolean value to another one, $\top$ is assigned, which signals a write conflict. Hence, this definition exactly models the intended behaviour.

$$\texttt{executeActionF\_def} \vdash_{\text{def}}$$
$$\mathsf{execAct}_{\mathbb{F}}(\mathcal{E}, a) = \mathcal{E}^{\mathsf{sup}_{\mathbb{F}}([\![\mathsf{lhs}(a)]\!]^{\mathbb{F}}_{\mathcal{E}}, [\![\mathsf{rhs}(a)]\!]^{\mathbb{F}}_{\mathcal{E}})}_{\mathsf{lhs}(a)}$$
$$\texttt{executeActionsF\_def} \vdash_{\text{def}}$$
$$(\mathsf{execActs}_{\mathbb{F}}(\mathcal{E}, \langle\rangle) = \mathcal{E}) \wedge$$
$$(\mathsf{execActs}_{\mathbb{F}}(\mathcal{E}, a :: \mathcal{A}) = \mathsf{execAct}_{\mathbb{F}}(\mathsf{execActs}_{\mathbb{F}}(\mathcal{E}, \mathcal{A}), a))$$

If no action is activated, the new value of a variable x is set to a default value, which is commonly referred to as *reaction to absence*. The function $\mathsf{reactToAbsense}_{\mathbb{F}}(\mathcal{E}, x, \mathcal{A})$ checks for a given output variable x, whether there is a possibly enabled action in $\mathcal{A}$ for x. If this is not the case, x is assigned its default value in environment $\mathcal{E}$.

$$\texttt{reactToAbsenseF\_def} \vdash_{\text{def}}$$
$$(\mathsf{reactToAbsense}_{\mathbb{F}}(\mathcal{E}, x, \langle\rangle) = \mathcal{E}^{\mathsf{default}(x)}_{x}) \wedge$$
$$(\mathsf{reactToAbsense}_{\mathbb{F}}(\mathcal{E}, x, a :: \mathcal{A}) =$$
$$\quad \textbf{if } x = \mathsf{lhs}(a) \textbf{ then } \mathcal{E} \textbf{ else } \mathsf{reactToAbsense}_{\mathbb{F}}(\mathcal{E}, x, \mathcal{A}))$$
$$\texttt{reactToAbsensesF\_def} \vdash_{\text{def}}$$
$$(\mathsf{reactToAbsenses}_{\mathbb{F}}(\mathcal{E}, \langle\rangle, \mathcal{A}) = \mathcal{E}) \wedge$$
$$(\mathsf{reactToAbsenses}_{\mathbb{F}}(\mathcal{E}, m :: \mathcal{V}, \mathcal{A}) =$$
$$\quad \mathsf{reactToAbsense}_{\mathbb{F}}(\mathsf{reactToAbsenses}_{\mathbb{F}}(\mathcal{E}, \mathcal{V}, \mathcal{A}), x, \mathcal{A})$$

Each step of the causality analysis consists in executing all must-actions and reacting to absence for all variables that have no can-actions. Note that must-actions of the previous iteration step are executed again in the next iteration. This must be done, since the expression on the right-hand side might not have been known. So, its value can change during the iterations.

$$\texttt{cmAnalysisStep\_def} \vdash_{\text{def}} \mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}) =$$
$$\mathsf{reactToAbsenses}_{\mathbb{F}}(\mathsf{execActs}_{\mathbb{F}}(\mathcal{E}, \mathsf{mustActs}_{\mathcal{E}}(\mathcal{A})), \mathcal{V}, \mathsf{canActs}_{\mathcal{E}}(\mathcal{A}))$$
$$\texttt{cmAnalysis\_dfn} \vdash_{\text{def}} \mathsf{cmAnalysis}(\mathcal{A}, \mathcal{V}, \mathcal{E}) =$$
$$\quad \textbf{let } \mathcal{E}' = \mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}) \textbf{ in}$$
$$\quad\quad \textbf{if } \mathcal{E} = \mathcal{E}' \textbf{ then } \mathcal{E} \textbf{ else } \mathsf{cmAnalysis}(\mathcal{A}, \mathcal{V}, \mathcal{E}')$$

To prove the termination of the fixpoint iteration performed by the can-must analysis $\mathsf{cmAnalysis}(\mathcal{A}, \mathcal{V}, \mathcal{E})$, we define a weight $\mathsf{envWeight}(\mathcal{E}, \mathcal{V})$ for an environment $\mathcal{E}$ with output variables $\mathcal{V}$. The weight of a variable intuitively reflects the amount of knowledge it stores, and the weight of an environment is just the sum of the weights of all output variables. Since the set of output variables $\mathcal{V}$ is finite, it has the upper bound $2|\mathcal{V}|$.

$$\texttt{weight4\_def} \vdash_{\text{def}}$$
$$(\mathsf{wt}(\bot) = 0) \wedge (\mathsf{wt}(0) = 1) \wedge (\mathsf{wt}(1) = 1) \wedge (\mathsf{wt}(\top) = 2)$$
$$\texttt{envWeight\_def} \vdash_{\text{def}}$$
$$(\mathsf{envWeight}(\mathcal{E}, \langle\rangle) = 0) \wedge$$
$$(\mathsf{envWeight}(\mathcal{E}, v :: \mathcal{V}) = \mathsf{envWeight}(\mathcal{E}, \mathcal{V}) + \mathsf{wt}(\mathcal{E}(v)))$$

Since the termination is apparent for the case that no action is executed and the environment remains the same, it remains to prove that each call to $\mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E})$ is monotonic in that it increases the weight of the environment. The proof basically follows the argument that the executions of all actions and the reactions to absence are monotonic. A variable with a Boolean value is never reset to $\bot$, and a $\top$ is never replaced by any other value. The proof is done by induction on the variables, followed by an induction on the actions. However, some more subtle preconditions are required for a successful proof, which are due to the general assumptions of the theory:

- Since the list of variables $\mathcal{V}$ models rather a set than an actual list, all of its elements must be distinct, so that the weight can be computed correctly.
- $\mathsf{varsDefined}(\mathcal{E}, \mathcal{V})$ assures that values for all output variables $\mathcal{V}$ can be found in the environment $\mathcal{E}$. Otherwise, their weights would be undefined. Alternatively, the weight of undefined variables could be defined, which would not reflect the real situation.
- The actions $\mathcal{A}$ must only modify the variables given by output variables $\mathcal{V}$. Otherwise, some updates would not affect the weight computation. Note that the set of output variables cannot be determined from the set of actions, since there may be some variables whose behaviour is only given by the reaction to absence.
- Finally, as already noted, the default value $\mathsf{default}(x)$ should be independent of the current environment. Otherwise, the reaction to absence would be able to cause additional dependencies.

$$
\begin{aligned}
&\texttt{varDefined\_def} \vdash_{\mathrm{def}} \mathsf{varDefined}(\mathcal{E}, v) = v \in \mathbf{domain}(\mathcal{E}) \\
&\texttt{varsDefined\_def} \vdash_{\mathrm{def}} \\
&\quad (\mathsf{varsDefined}(\mathcal{E}, \langle\rangle) = \mathsf{true}) \wedge \\
&\quad (\mathsf{varsDefined}(\mathcal{E}, v :: \mathcal{V}) = \mathsf{varDefined}(\mathcal{E}, v) \wedge \mathsf{varsDefined}(\mathcal{E}, \mathcal{V})) \\
&\texttt{actionValid\_def} \vdash_{\mathrm{def}} \mathsf{actionValid}(\mathcal{V}, a) = \mathsf{lhs}(a) \in \mathcal{V} \\
&\texttt{actionsValid\_def} \vdash_{\mathrm{def}} \\
&\quad (\mathsf{actionsValid}(\mathcal{V}, \langle\rangle) = \mathsf{true}) \wedge \\
&\quad (\mathsf{actionsValid}(\mathcal{V}, a :: \mathcal{A}) = \mathsf{actionValid}(\mathcal{V}, a) \wedge \mathsf{actionsValid}(\mathcal{V}, \mathcal{A})) \\
&\texttt{STEP\_MONOTONE} \vdash \\
&\quad \mathsf{distinct}(\mathcal{V}) \rightarrow \mathsf{varsDefined}(\mathcal{E}, \mathcal{V}) \rightarrow \mathsf{actionsValid}(\mathcal{V}, \mathcal{A}) \rightarrow \\
&\quad (\forall \mathcal{E}_1\, \mathcal{E}_2\, x.\, (\llbracket \mathsf{default}(x) \rrbracket^{\mathbb{F}}_{\mathcal{E}_1}) = (\llbracket \mathsf{default}(x) \rrbracket^{\mathbb{F}}_{\mathcal{E}_2})) \rightarrow \\
&\qquad (\mathsf{envWeight}(\mathcal{V}, \mathcal{E}) \leq \mathsf{envWeight}(\mathcal{V}, \mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E})))
\end{aligned}
$$

This concludes the formalisation of the can-must analysis, which serves as a specification for other algorithms for causality analysis.

3 Causality Analysis by Model Checking

In this section, we describe our symbolic approach to causality analysis, which is based on a symbolic description of a transition system, so that the causality analysis can be formulated as a model checking problem. In the following, we describe the transition system, before we explain the actual verification task in Subsection 3.2.

3.1 Modelling the Progress of Information

To encode the causality analysis as a model checking problem, we must create a transition system that explicitly models the progress of information and the occurrence of write conflicts. Therefore, in addition to the variables in the program, we introduce for every output variable x a Boolean-typed variable x^{kn} that holds iff the value of x is known. If this is the case, then the value of x is stored in the variable x, otherwise the value of x is not yet known and we have to ignore its content. Similarly, a second Boolean-typed variable x^{wc} is added that holds iff a write conflict occurred for variable x. If this bit is set, we can also ignore the content of the corresponding variable.

Using the variables x^{kn} and x^{wc}, 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.

As a first step, we define a function that maps a program expression σ to a Boolean formula $\mathsf{wc}(\sigma)$ such that $\mathsf{wc}(\sigma)$ holds iff the expression σ cannot be evaluated due to a write conflict. Basically, this is always the case if a subformula cannot be evaluated:

- For variables and constants, we define:
  - $\mathsf{wc}(x) := \begin{cases} \mathsf{false} & \text{: if } x \text{ is an input variable} \\ x^{\mathsf{wc}} & \text{: otherwise} \end{cases}$
  - $\mathsf{wc}(c) := \mathsf{false}$
- For the Boolean operators, we define:
  - $\mathsf{wc}(\mathit{not}(\varphi)) := \mathsf{wc}(\varphi)$
  - $\mathsf{wc}(\mathit{and}(\varphi, \psi)) := \mathsf{wc}(\varphi) \vee \mathsf{wc}(\psi)$
  - $\mathsf{wc}(\mathit{or}(\varphi, \psi)) := \mathsf{wc}(\varphi) \vee \mathsf{wc}(\psi)$

Analogously, we formally define a function that maps a program expression σ to a Boolean formula $\mathsf{kn}(\sigma)$ such that $\mathsf{kn}(\sigma)$ holds if and only if the expression σ can be evaluated to a known value. The formula $\mathsf{kn}(\sigma)$ encodes the lazy evaluation rules of the Boolean operators, which are shown in Figure 3.

- For variables and constants, we define:
  - $\mathsf{kn}(x) := \begin{cases} \mathsf{true} & \text{: if x is an input variable} \\ x^{\mathsf{kn}} & \text{: otherwise} \end{cases}$
  - $\mathsf{kn}(c) := \mathsf{true}$
- For the Boolean operators, we define:
  - $\mathsf{kn}(\mathit{not}(\tau_1)) := \mathsf{wc}(\tau_1) \vee \mathsf{kn}(\tau_1)$

$x \wedge \mathsf{false} = \mathsf{false}$	$\mathsf{false} \wedge x = \mathsf{false}$
$x \vee \mathsf{true} = \mathsf{true}$	$\mathsf{true} \vee x = \mathsf{true}$
$\mathsf{false} \rightarrow x = \mathsf{true}$	$x \rightarrow \mathsf{true} = \mathsf{true}$

Fig. 3. Lazy Evaluation Rules

- $\mathsf{kn}(\mathit{and}(\tau_1, \tau_2)) := \mathsf{wc}(\tau_1) \vee \mathsf{wc}(\tau_2) \vee \mathsf{kn}(\tau_1) \wedge \mathsf{kn}(\tau_2) \vee \mathsf{kn}(\tau_1) \wedge (\tau_1 = \mathsf{true}) \vee \mathsf{kn}(\tau_2) \wedge (\tau_2 = \mathsf{true})$
- $\mathsf{kn}(\mathit{or}(\tau_1, \tau_2)) := \mathsf{wc}(\tau_1) \vee \mathsf{wc}(\tau_2) \vee \mathsf{kn}(\tau_1) \wedge \mathsf{kn}(\tau_2) \vee \mathsf{kn}(\tau_1) \wedge (\tau_1 = \mathsf{false}) \vee \mathsf{kn}(\tau_2) \wedge (\tau_2 = \mathsf{false})$

Clearly, in the actual behaviour 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 all output variables and try to determine their values with the micro steps that can occur in a macro step. The procedure is repeated for each reaction.

For this reason, we introduce a clock signal tick, which is true whenever all variables have become known values, and a new reaction may be started. If this happens, the system state may change according to the control flow, and the input variables are allowed to change their values in a non-deterministic way (which reflects the uncontrollable input of the environment).

Propagation of Knowledge:

$$\mathsf{KnownTrans}_x :\equiv \left(\mathsf{next}(x^{\mathsf{kn}}) :\Leftrightarrow \left(\begin{array}{l} \neg\mathsf{tick} \wedge \mathsf{kn}(x) \vee \\ \neg\mathsf{tick} \wedge \left(\bigvee_{j=1}^{p} \mathsf{kn}(\gamma_j) \wedge \gamma_j \wedge \mathsf{kn}(\tau_j) \right) \vee \\ \neg\mathsf{tick} \wedge \left(\bigwedge_{j=1}^{p} \mathsf{kn}(\gamma_j) \wedge \neg\gamma_j \right) \end{array} \right) \right)$$

Propagation of Write Conflicts:

$$\mathsf{WCTrans}_x :\equiv \left(\mathsf{next}(x^{\mathsf{wc}}) :\Leftrightarrow \left(\begin{array}{l} \neg\mathsf{tick} \wedge \mathsf{wc}(x) \vee \\ \neg\mathsf{tick} \wedge \mathsf{kn}(x) \wedge \\ \qquad \left(\bigvee_{j=1}^{p} \mathsf{kn}(\gamma_j) \wedge \gamma_j \wedge \mathsf{kn}(\tau_j) \wedge (x \neq \tau_j) \right) \vee \\ \neg\mathsf{tick} \wedge \\ \qquad \left(\begin{array}{l} \bigvee_{j=1}^{p-1} \bigvee_{k=j+1}^{p} \mathsf{kn}(\gamma_j) \wedge \gamma_j \wedge \mathsf{kn}(\tau_j) \wedge \\ \quad \mathsf{kn}(\gamma_k) \wedge \gamma_k \wedge \mathsf{kn}(\tau_k) \wedge (\tau_j \neq \tau_k) \end{array} \right) \end{array} \right) \right)$$

Computation of Values:

$$\mathsf{ValTrans}_x :\equiv \left(\begin{array}{l} \left(\neg\mathsf{tick} \rightarrow \left(\bigwedge_{j=1}^{p} \neg\mathsf{kn}(x) \wedge \mathsf{kn}(\gamma_j) \wedge \gamma_j \wedge \mathsf{kn}(\tau_j) \rightarrow \mathsf{next}(x) = \tau_j \right) \right) \wedge \\ \left(\neg\mathsf{tick} \rightarrow \left(\mathsf{kn}(x) \vee \neg \left(\bigvee_{j=1}^{p} \mathsf{kn}(\gamma_j) \wedge \gamma_j \wedge \mathsf{kn}(\tau_j) \right) \rightarrow \mathsf{next}(x) = x \right) \right) \wedge \\ \left(\mathsf{tick} \rightarrow (\mathsf{next}(x) = \mathsf{default}(x)) \right) \end{array} \right)$$

Fig. 4. Causality Transition Relation of Variable x

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 variables $\mathsf{kn}(x) = x^{\mathsf{kn}}$ and $\mathsf{wc}(x) = x^{\mathsf{wc}}$.

The transition relation of the information flow can be formulated as an equation system as shown in Figure 4 that contains the following cases for x^{kn}:

- If there is no clock tick, the value of x remains known if it was already known.
- If there is no clock tick, the value of x becomes known if a guarded action $(\gamma_j, \mathtt{x}{=}\tau_j)$ with an assignment $\mathtt{x}{=}\tau_j$ 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 4 simply demands that $\mathsf{next}(\mathrm{x})=\mathrm{x}$ has to hold in this case, since x has been given the now desired value at the first step of the reaction as a preliminary value.
- Otherwise, the value of x is not known.

Write conflicts are propagated according to the following rules:

- If there is no clock tick, a write conflict for x persists.
- If there is no clock tick, a write conflict occurs, if a guarded action $(\gamma_j, \mathtt{x}{=}\tau_j)$ is activated that assigns a different value to an already known variable.
- If there is no clock tick, a write conflict occurs, if there are two guarded actions that are activated and assign different values to the same variable.

The data flow of x is determined by the same cases as formalised in formula $\mathsf{ValTrans}_x$ given in Figure 4:

- If there is no clock tick and a guarded action $(\gamma_j, \mathtt{x}{=}\tau_j)$ with an assignment $\mathtt{x}{=}\tau_j$ can be fired, then x will receive the value of τ_j at the next point of time.
- If there is no clock tick and no guarded action can be fired, then x will keep its value. This covers two 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. Second, if a write conflict occurs, the value of x is irrelevant.
- 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, x is initialised to its default value $\mathsf{default}(x)$.

The formulas given in Figure 4 describe the transition relation of a particular variable x. The complete transition relation is therefore the conjunction of the partial transition relations of all output variables. In addition to this, we also have to determine, when the causality analysis terminates. This is accomplished by the clock signal, which is defined as follows:

$$\mathsf{ClockTrans} :\equiv \left(\mathsf{tick} :\Leftrightarrow \bigwedge_x \mathsf{kn}(x) \wedge \neg \mathsf{wc}(x) \right)$$

The new clock tick can arrive as soon as all values of the local and output variables become known. New input variables can then be read for the next reaction.

3.2 Model Checking Tasks

It is easily seen that the micro step behaviour as formalised in the previous subsection describes the semantics of an *asynchronous circuit.* Obviously, a program is causally correct, if the values of all variables can be determined for all possible inputs. Due to the definition of ClockTrans, the states in which all variables are known can be identified by the tick variable, which also marks the beginning of a new reaction. Hence, a single reaction is modelled by a finite chain of transitions in this 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 no lead to a valid new state, i.e. tick does not appear after the initial step. Such execution sequences end in a self-loop of a state without a tick. Thus, any system that is guaranteed to hit a $clock$ state after the initialisation, is causally correct. This property can be simply denoted by XF tick in linear time temporal logic and can be checked by any state-of-the-art model checker that supports linear time logic (LTL).

4 Formalisation of Alternative Analysis

Traditional causality analysis differs from our symbolic procedure in two aspects: first, the data is represented in a different way. Instead of four-valued truth values, the symbolic formulation is based on the original data types plus two additional status bits (to allow the use of state-of-the-art model checkers). Second, the transition relation has a denotational style, while traditional causality analysis describes the fixpoint computation operationally. Our formalisation and the equivalence proof of the equivalence are therefore divided into two parts. The next two sections first link both data representations, while Section4.3 bridges the second gap.

4.1 Two-Valued Environment

Similar to the definitions of the four-valued environment, we hide the functions to read and store values in the environment behind the notations $\mathcal{E}(x)$ and $\mathcal{E}^v_x$, respectively. Analogously, the additional status bits for knowledge and write conflicts can be accessed by $\mathcal{E}(x^{\mathsf{kn}})$, $\mathcal{E}^v_{x^{\mathsf{kn}}}$, $\mathcal{E}(x^{\mathsf{wc}})$ and $\mathcal{E}^v_{x^{\mathsf{wc}}}$, respectively.

The fundamental definition to link both variants of causality analysis, which is the basis for the equivalence proof, is given by $\mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2)$. This relation takes two environments, a four-valued one and a two-valued one and defines whether they are equivalent or not. For this task, it makes a case distinction on the four possible truth values of the can-must analysis:

- If a four-valued variable is $\bot$, the two-valued counterpart should be marked as not known and without write conflicts.
- If the value of variable is 0 (or 1), it should be marked as known with no write conflict and its value is set to 0 (or 1, respectively).
- If the value of the variable is $\top$, it should be marked as known with a write conflict.
- The remaining case for the two-valued environment, i. e. a variable is not known and has a write conflict, is forbidden.

The following definition respects these considerations:

$$\begin{aligned}
&\texttt{envEqual_def} \vdash_{\text{def}} \mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2) = \\
&\quad (\forall x.(\mathcal{E}_1(x) = \bot) = \neg\mathcal{E}_2(x^{\mathsf{kn}}) \wedge \neg\mathcal{E}_2(x^{\mathsf{wc}})) \wedge \\
&\quad (\forall x.(\mathcal{E}_1(x) = \mathsf{0}) = \mathcal{E}_2(x^{\mathsf{kn}}) \wedge \neg\mathcal{E}_2(x^{\mathsf{wc}}) \wedge (\mathcal{E}_2(x) = \mathsf{0})) \wedge \\
&\quad (\forall x.(\mathcal{E}_1(x) = \mathsf{1}) = \mathcal{E}_2(x^{\mathsf{kn}}) \wedge \neg\mathcal{E}_2(x^{\mathsf{wc}}) \wedge (\mathcal{E}_2(x) = \mathsf{1})) \wedge \\
&\quad (\forall x.(\mathcal{E}_1(x) = \top) = \mathcal{E}_2(x^{\mathsf{kn}}) \wedge \mathcal{E}_2(x^{\mathsf{wc}})) \wedge \\
&\qquad (\forall x.\ \mathsf{false} = \neg\mathcal{E}_2(x^{\mathsf{kn}}) \wedge \mathcal{E}_2(x^{\mathsf{wc}})
\end{aligned}$$

On this basis, the evaluation $[\![e]\!]_{\mathcal{E}}$ of the Boolean expressions is defined:

$$\begin{aligned}
&\texttt{blEval_def} \vdash_{\text{def}} \\
&\quad ([\![\mathit{false}]\!]_{\mathcal{E}} = \mathsf{false}) \wedge ([\![\mathit{true}]\!]_{\mathcal{E}} = \mathsf{true}) \wedge \\
&\quad ([\![x]\!]_{\mathcal{E}} = \mathcal{E}(x)) \wedge \\
&\quad ([\![\mathit{not}(\tau)]\!]_{\mathcal{E}} = \neg[\![\tau]\!]_{\mathcal{E}}) \wedge \\
&\quad ([\![\mathit{and}(\tau_1, \tau_2)]\!]_{\mathcal{E}} = [\![\tau_1]\!]_{\mathcal{E}} \wedge [\![\tau_2]\!]_{\mathcal{E}}) \wedge \\
&\quad ([\![\mathit{or}(\tau_1, \tau_2)]\!]_{\mathcal{E}} = [\![\tau_1]\!]_{\mathcal{E}} \vee [\![\tau_2]\!]_{\mathcal{E}})
\end{aligned}$$

The definition for the write conflict status needs some considerations. Following the approach of the previous section, its determination should be strict. If a write conflict occurs in any subterm, it is propagated. This models the fact that a model that contains inconsistent variables is completely inconsistent.

$$\begin{aligned}
&\texttt{blWriteConflict_def} \vdash_{\text{def}} \\
&\quad (\mathsf{wc}_{\mathcal{E}}(\mathit{false}) = \mathsf{false}) \wedge (\mathsf{wc}_{\mathcal{E}}(\mathit{true}) = \mathsf{false}) \wedge \\
&\quad (\mathsf{wc}_{\mathcal{E}}(x) = \mathcal{E}(x^{\mathsf{wc}})) \wedge \\
&\quad (\mathsf{wc}_{\mathcal{E}}(\mathit{not}(\tau)) = \mathsf{wc}_{\mathcal{E}}(\tau)) \wedge \\
&\quad (\mathsf{wc}_{\mathcal{E}}(\mathit{and}(\tau_1, \tau_2)) = \mathsf{wc}_{\mathcal{E}}(\tau_1) \vee \mathsf{wc}_{\mathcal{E}}(\tau_2)) \wedge \\
&\quad (\mathsf{wc}_{\mathcal{E}}(\mathit{or}(\tau_1, \tau_2)) = \mathsf{wc}_{\mathcal{E}}(\tau_1) \vee \mathsf{wc}_{\mathcal{E}}(\tau_2))
\end{aligned}$$

In contrast to this, the known status makes use of lazy evaluation.

$$\begin{aligned}
&\texttt{blKnown_def} \vdash_{\text{def}} \\
&\quad (\mathsf{kn}_{\mathcal{E}}(\mathit{false}) = \mathsf{true}) \wedge (\mathsf{kn}_{\mathcal{E}}(\mathit{true}) = \mathsf{true}) \wedge \\
&\quad (\mathsf{kn}_{\mathcal{E}}(x) = \mathcal{E}(x^{\mathsf{kn}})) \wedge \\
&\quad (\mathsf{kn}_{\mathcal{E}}(\mathit{not}(\tau)) = \mathsf{wc}_{\mathcal{E}}(\tau) \vee \mathsf{kn}_{\mathcal{E}}(\tau)) \wedge \\
&\quad (\mathsf{kn}_{\mathcal{E}}(\mathit{and}(\tau_1, \tau_2)) = \\
&\qquad \mathsf{wc}_{\mathcal{E}}(\tau_1) \vee \mathsf{wc}_{\mathcal{E}}(\tau_2) \vee (\mathsf{kn}_{\mathcal{E}}(\tau_1) \wedge \mathsf{kn}_{\mathcal{E}}(\tau_2)) \vee \\
&\qquad (\mathsf{kn}_{\mathcal{E}}(\tau_1) \wedge ([\![\tau_1]\!]_{\mathcal{E}} = \mathsf{false})) \vee (\mathsf{kn}_{\mathcal{E}}(\tau_2) \wedge ([\![\tau_1]\!]_{\mathcal{E}} = \mathsf{false}))) \wedge \\
&\quad (\mathsf{kn}_{\mathcal{E}}(\mathit{or}(\tau_1, \tau_2)) = \\
&\qquad \mathsf{wc}_{\mathcal{E}}(\tau_1) \vee \mathsf{wc}_{\mathcal{E}}(\tau_2) \vee (\mathsf{kn}_{\mathcal{E}}(\tau_1) \wedge \mathsf{kn}_{\mathcal{E}}(\tau_2)) \vee \\
&\qquad (\mathsf{kn}_{\mathcal{E}}(\tau_1) \wedge ([\![\tau_1]\!]_{\mathcal{E}} = \mathsf{true})) \vee (\mathsf{kn}_{\mathcal{E}}(\tau_2) \wedge ([\![\tau_1]\!]_{\mathcal{E}} = \mathsf{true})))
\end{aligned}$$

The first proof obligation is that the previous definitions comply with the four-valued ones. This corresponds to a lifting of the environments for the variables to the expressions, i. e. : provided that variables in the current environments are considered equal, all expressions can be considered to be equal, too. This goal can be proved with the calculation rules for the four-valued operations, the given definitions and a first-order tactic automatically, after initiating an induction on the structure of the expressions.

$$\begin{aligned}
&\texttt{EXPR_EQUAL} \vdash \mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2) \rightarrow \\
&\quad ((\llbracket\tau\rrbracket^{\mathbb{F}}_{\mathcal{E}_1} = \bot) = \neg\mathsf{kn}_{\mathcal{E}_2}(\tau) \wedge \neg\mathsf{wc}_{\mathcal{E}_2}(\tau)) \wedge \\
&\quad ((\llbracket\tau\rrbracket^{\mathbb{F}}_{\mathcal{E}_1} = 0) = \mathsf{kn}_{\mathcal{E}_2}(\tau) \wedge \neg\mathsf{wc}_{\mathcal{E}_2}(\tau) \wedge (\llbracket\tau_1\rrbracket_{\mathcal{E}_2} = \mathsf{false})) \wedge \\
&\quad ((\llbracket\tau\rrbracket^{\mathbb{F}}_{\mathcal{E}_1} = 1) = \mathsf{kn}_{\mathcal{E}_2}(\tau) \wedge \neg\mathsf{wc}_{\mathcal{E}_2}(\tau) \wedge (\llbracket\tau_1\rrbracket_{\mathcal{E}_2} = \mathsf{true})) \wedge \\
&\quad ((\llbracket\tau\rrbracket^{\mathbb{F}}_{\mathcal{E}_1} = \top) = \mathsf{kn}_{\mathcal{E}_2}(\tau) \wedge \mathsf{wc}_{\mathcal{E}_2}(\tau)) \wedge \\
&\quad (\neg(\neg\mathsf{kn}_{\mathcal{E}_2}(\tau) \wedge \mathsf{wc}_{\mathcal{E}_2}(\tau)))
\end{aligned}$$

This proof is done by structural induction on expressions. Surprisingly, it revealed some glitches in the definitions in former versions of $\mathsf{kn}_{\mathcal{E}}(\tau)$, which did not respect some write conflicts.

4.2 Execution of Actions

The next step to define the execution of actions and to prove is that both variants perform exactly the same steps, each one in its representation. Provided that the environments have been equivalent before the execution of an action, they must be equivalent after the execution. This is assured by the following definitions and theorems:

$$\begin{aligned}
&\texttt{executeGuardedAction4_def} \vdash_{\mathrm{def}} \mathsf{executeGuardedAction}_{\mathbb{F}}(a, \mathcal{E}) = \\
&\quad \textbf{if } \mathsf{must}(\llbracket\mathsf{grd}(a)\rrbracket_{\mathcal{E}_{\mathbb{F}}}) \textbf{ then } \mathsf{execAct}_{\mathbb{F}}(\mathcal{E}_{\mathbb{F}}, a) \textbf{ else } \mathcal{E}_{\mathbb{F}} \\
&\texttt{actionExecutable_def} \vdash_{\mathrm{def}} \mathsf{actionExecutable}(\mathcal{E}, a) = \\
&\quad \mathsf{kn}_{\mathcal{E}}(\mathsf{grd}(a)) \wedge (\llbracket\mathsf{grd}(a)\rrbracket_{\mathcal{E}} = \mathsf{true}) \wedge \mathsf{kn}_{\mathcal{E}}(\mathsf{rhs}(a)) \\
&\texttt{actionConflictFree_def} \vdash_{\mathrm{def}} \mathsf{actionConflictFree}(\mathcal{E}, a) = \\
&\quad \neg\mathsf{kn}_{\mathcal{E}}(\mathsf{lhs}(a)) \vee (\llbracket\mathsf{lhs}(a)\rrbracket_{\mathcal{E}} = \llbracket\mathsf{rhs}(a)\rrbracket_{\mathcal{E}}) \\
&\texttt{executeGuardedAction_def} \vdash_{\mathrm{def}} \mathsf{executeGuardedAction}(a, \mathcal{E}) = \\
&\quad \textbf{if } \mathsf{actionExecutable}(\mathcal{E}, a) \textbf{ then} \\
&\qquad (\textbf{if } \mathsf{actionConflictFree}(\mathcal{E}, a) \textbf{ then } \mathsf{execAct}_{\mathbb{F}}(\mathcal{E}, a) \textbf{ else } \mathcal{E}^{1}_{a^{\mathsf{wc}}})^{1}_{a^{\mathsf{kn}}} \\
&\quad \textbf{else } \mathcal{E} \\
&\texttt{ACTION_EQUAL} \vdash \mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2) \rightarrow \\
&\quad \mathsf{envEqual}(\mathsf{executeGuardedAction}_{\mathbb{F}}(a, \mathcal{E}_1), \mathsf{executeGuardedAction}(a, \mathcal{E}_2))
\end{aligned}$$

The proof of the last theorem basically makes a case distinction on the following situations given by the value and status of the expressions occurring in a guarded action a:

- The guard $\mathsf{grd}(a)$ is not known or it is false. In both cases, the value of the left-hand side is not changed by the execution of the guarded action, since the **else** branches in both representations are taken.
- The guard $\mathsf{grd}(a)$ is known and not false, and the right-hand side expression $\mathsf{rhs}(a)$ is unknown. The value of the left-hand side is not changed in both environment, due to the following reasons. In the environment $\mathcal{E}$, the whole action a is not executed. In the environment $\mathcal{E}_{\mathbb{F}}$, the action is executed, in principle. However, it does not have any effect, since the maximum of the old value and the value of the unknown right-hand side (which is $\bot$ by assumption) is always the old value.
- The guard of the guarded action is known and not false. The right-hand side expression is known. The environment is updated, where the update depends whether a write conflict is caused or not. If this is not the case, the value of the left-hand side is updated and set to be known. Otherwise, its write conflict status is set in both representations.

$$
\begin{array}{l}
\texttt{actionActive\_def} \vdash_{\text{def}} \\
\quad \mathsf{active}_{\mathcal{E}}(a) = \mathsf{kn}_{\mathcal{E}}(\mathsf{grd}(a)) \wedge [\![\mathsf{grd}(a)]\!]_{\mathcal{E}} \wedge \mathsf{kn}_{\mathcal{E}}(\mathsf{rhs}(a)) \\
\texttt{someActionActive\_def} \vdash_{\text{def}} \\
\quad (\mathsf{someActive}_{\mathcal{E}}(\langle\rangle) = \mathsf{false}) \wedge \\
\quad (\mathsf{someActive}_{\mathcal{E}}(a :: \mathcal{A}) = \mathsf{active}_{\mathcal{E}}(a) \vee \mathsf{someActive}_{\mathcal{E}}(\mathcal{A})) \\
\texttt{allActionsInactive\_def} \vdash_{\text{def}} \\
\quad (\mathsf{allInactive}_{\mathcal{E}}(\langle\rangle) = \mathsf{true}) \wedge \\
\quad (\mathsf{allInactive}_{\mathcal{E}}(a :: \mathcal{A}) = \neg\mathsf{active}_{\mathcal{E}}(a) \wedge \mathsf{allInactive}_{\mathcal{E}}(\mathcal{A})) \\
\texttt{conflictingActionActive\_def} \vdash_{\text{def}} \\
\quad \mathsf{conflActive}_{\mathcal{E}}(a) = \mathsf{active}_{\mathcal{E}}(a) \wedge \mathsf{kn}_{\mathcal{E}}(\mathsf{lhs}(a)) \wedge ([\![\mathsf{lhs}(a)]\!]_{\mathcal{E}} \neq [\![\mathsf{rhs}(a)]\!]_{\mathcal{E}}) \\
\texttt{conflictingActionsActive\_def} \vdash_{\text{def}} \\
\quad (\mathsf{conflsActive}_{\mathcal{E}}(\langle\rangle) = \mathsf{false}) \wedge \\
\quad (\mathsf{conflsActive}_{\mathcal{E}}(a :: \mathcal{A}) = \mathsf{conflActive}_{\mathcal{E}}(a) \vee \mathsf{conflsActive}_{\mathcal{E}}(\mathcal{A})) \\
\texttt{inconsistentActionActive\_def} \vdash_{\text{def}} \\
\quad (\mathsf{inconActive}_{\mathcal{E}}(a_0, \langle\rangle) = \mathsf{false}) \\
\quad (\mathsf{inconActive}_{\mathcal{E}}(a_0, a_1 :: \mathcal{A}) = \mathsf{inconActive}_{\mathcal{E}}(a_0, \mathcal{A}) \vee \\
\qquad \mathsf{active}_{\mathcal{E}}(a_0) \wedge \mathsf{active}_{\mathcal{E}}(a_1) \wedge ([\![\mathsf{rhs}(a_0)]\!]_{\mathcal{E}} \neq [\![\mathsf{rhs}(a_1)]\!]_{\mathcal{E}})) \\
\texttt{inconsistentActionsActive\_def} \vdash_{\text{def}} \\
\quad (\mathsf{inconsActive}_{\mathcal{E}}(\langle\rangle) = \mathsf{false}) \\
\quad (\mathsf{inconsActive}_{\mathcal{E}}(a :: \mathcal{A}) = \mathsf{inconActive}_{\mathcal{E}}(a, \mathcal{A}) \vee \mathsf{inconsActive}_{\mathcal{E}}(\mathcal{A})) \\
\\
\texttt{trWCVar\_def} \vdash_{\text{def}} \mathsf{trWCVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}') = \\
\quad (\mathsf{wc}_{\mathcal{E}'}(v) = \mathsf{wc}_{\mathcal{E}}(v) \vee \mathsf{conflActive}_{\mathcal{E}}(\mathcal{A}_v) \vee \mathsf{inconsActive}_{\mathcal{E}}(\mathcal{A}_v) \\
\texttt{trWC\_def} \vdash_{\text{def}} \mathsf{trWC}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') = \bigwedge_{v \in \mathcal{V}} \mathsf{trWCVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}') \\
\\
\texttt{trKnVar\_def} \vdash_{\text{def}} \mathsf{trKnVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}') = \\
\quad (\mathsf{kn}_{\mathcal{E}'}(v) = \mathsf{kn}_{\mathcal{E}}(v) \vee \mathsf{someActive}_{\mathcal{E}}(\mathcal{A}_v) \vee \mathsf{allInactive}_{\mathcal{E}}(\mathcal{A}_v)) \\
\texttt{trKn\_def} \vdash_{\text{def}} \mathsf{trKn}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') = \bigwedge_{v \in \mathcal{V}} \mathsf{trKnVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}')) \\
\\
\texttt{trValAct\_def} \vdash_{\text{def}} \mathsf{trValAct}(a, v, \mathcal{E}, \mathcal{E}') = \\
\quad \neg\mathsf{kn}_{\mathcal{E}}(v) \wedge \mathsf{active}_{\mathcal{E}}(a) \rightarrow ([\![v]\!]_{\mathcal{E}'} = [\![\mathsf{rhs}(a)]\!]_{\mathcal{E}}) \\
\texttt{trValActs\_def} \vdash_{\text{def}} \\
\quad (\mathsf{trValActs}(\langle\rangle, v, \mathcal{E}, \mathcal{E}') = \mathsf{true}) \wedge \\
\quad (\mathsf{trValActs}(a :: \mathcal{A}, v, \mathcal{E}, \mathcal{E}') = \mathsf{trValActs}(a, v, \mathcal{E}, \mathcal{E}') \wedge \mathsf{trValAct}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}')) \\
\texttt{trValVar\_def} \vdash_{\text{def}} \mathsf{trValVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}') = \\
\quad \mathsf{trValActs}(\mathcal{A}_v, v, \mathcal{E}, \mathcal{E}') \wedge (\mathsf{kn}_{\mathcal{E}}(v) \vee \mathsf{allInactive}_{\mathcal{E}}(\mathcal{A}_v) \rightarrow ([\![v]\!]_{\mathcal{E}} = [\![v]\!]_{\mathcal{E}'})) \\
\texttt{trVal\_def} \vdash_{\text{def}} \mathsf{trVal}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') = \bigwedge_{v \in \mathcal{V}} \mathsf{trValVar}(\mathcal{A}, v, \mathcal{E}, \mathcal{E}') \\
\\
\texttt{trAnalysisStep\_def} \vdash_{\text{def}} \\
\quad \mathsf{trAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') = \\
\qquad \mathsf{trKn}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') \wedge \mathsf{trWC}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}') \wedge \mathsf{trVal}(\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{E}')
\end{array}
$$

Fig. 5. Formalisation of the Transition Relation

4.3 Transition Relation

A last step to link both variants of causality analysis remains, the description style: the previous paragraph still formalises the analysis operationally, it describes how to move from one state to another. In contrast, the transition relation of the model-checking

analysis has a different, rather denotational view. It describes the possible transitions and makes it possible that no behaviours or multiple behaviours exist (instead of a single one defined by the operational description). Hence, we have to prove that the transition relation exactly describes the steps that would be executed by an iteration in the operational description of the previous sections.

The complete formalisation of the transition relation is given in Figure 5. We closely follow the definitions for a single variable given in Section 3, but use some auxiliary definitions to keep the formalisation traceable and readable. Furthermore, we do not integrate the tick signal in our formalisation, but replace it by an initialisation of the output variables.

The following two theorems are the final step of our equivalence proof, which shows that a step in the transition relation corresponds to a step in the can-must analysis.

$$\begin{aligned}&\texttt{TRANSREL\_CORRECTNESS} \vdash \\ &\quad \mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2) \rightarrow \mathsf{envEqual}(\mathcal{E}_1', \mathcal{E}_2') \rightarrow \\ &\quad (\mathcal{E}_1' = \mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}_1)) \rightarrow \mathsf{trAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}_2, \mathcal{E}_2') \\ &\texttt{TRANSREL\_COMPLETENESS} \vdash \\ &\quad \mathsf{envEqual}(\mathcal{E}_1, \mathcal{E}_2) \rightarrow \mathsf{envEqual}(\mathcal{E}_1', \mathcal{E}_2') \rightarrow \\ &\quad \mathsf{trAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}_2, \mathcal{E}_2') \rightarrow (\mathcal{E}_1' = \mathsf{cmAnalysisStep}(\mathcal{A}, \mathcal{V}, \mathcal{E}_1))\end{aligned}$$

Critical points are the variables that are updated multiple times in the course of a step, e. g. write conflicts are typical examples for this. There, we must abstract from the intermediate environments, which come from the sequential execution of actions within a single iteration in the traditional can-must analysis. The equivalence proof uses the fact that they can be reordered, i. e. the execution of actions has the Church-Rosser property. Hence, a cumulated action with the same effect can be defined, which is subsequently shown to be equivalent with the transition relation.

5 Conclusions

In this paper, we have presented a new symbolic causality analysis based on model checking, which supports arbitrary data types and run-time error checking. With the help of the HOL4 theorem prover, we formalised our new approach as well as the traditional can-must analysis and showed their equivalence. Thus, we gained a formally verified symbolic causality analysis, which can be used in particular by compilers of synchronous languages.

References

1. Andersen, F.: A Theorem Prover for UNITY in Higher Order Logic. PhD thesis, Horsholm, Denmark (March 1992)
2. Andersen, F., Petersen, K.D., Petterson, J.S.: Program verification using HOL-UNITY. In: Joyce, J.J., Seger, C.-J.H. (eds.) HUG 1993. LNCS, vol. 780, pp. 1–15. Springer, Heidelberg (1994)
3. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE 91(1), 64–83 (2003)

4. Berry, G.: The constructive semantics of pure Esterel (July 1999), http://www-sop.inria.fr/esterel.org/
5. Brzozowski, J.A., Seger, C.-J.: Asynchronous Circuits. Springer, Heidelberg (1995)
6. Chandy, K.M., Misra, J.: Parallel Program Design, May 1989. Addison Wesley, Austin, Texas (1989)
7. Collins, G., Syme, D.: A theory of finite maps. In: Schubert, E.T., Alves-Foss, J., Windley, P. (eds.) HUG 1995. LNCS, vol. 971, pp. 122–137. Springer, Heidelberg (1995)
8. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 18(6), 742–760 (1999)
9. Huffman, D.: Combinational circuits with feedback. In: Mukhopadhyay, A. (ed.) Recent Developments in Switching Theory, pp. 27–55. Academic Press, London (1971)
10. Jantsch, A.: Modeling Embedded Systems and SoCs. Morgan Kaufmann, San Francisco (2004)
11. Kautz, W.: The necessity of closed circuit loops in minimal combinational circuits. IEEE Transactions on Computers C-19(2), 162–166 (1970)
12. Lee, E., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(12), 1217–1229 (1998)
13. Malik, S.: Analysis of cycle combinational circuits. IEEE Transactions on Computer Aided Design 13(7), 950–956 (1994)
14. Riedel, M.: Cyclic Combinational Circuits. PhD thesis, California Institute of Technology, Passadena, California (2004)
15. Riedel, M.D., Bruck, J.: Cyclic combinational circuits: Analysis for synthesis. In: International Workshop on Logic and Synthesis (IWLS), Laguna Beach, California (2003)
16. Riedel, M.D., Bruck, J.: The synthesis of cyclic combinational circuits. In: Design Automation Conference (DAC), Anaheim, California, USA, pp. 163–168. ACM Press, New York (2003)
17. Rivest, R.: The necessity of feedback in minimal monotone combinational circuits. IEEE Transactions on Computers C-26(6), 606–607 (1977)
18. Schneider, K.: The synchronous programming language Quartz. Internal Report, Department of Computer Science, University of Kaiserslautern (to appear, 2008)
19. Schneider, K., Brandt, J., Schuele, T.: Causality analysis of synchronous programs with delayed actions. In: Compilers, Architecture, and Synthesis for Embedded Systems (CASES), pp. 179–189. ACM Press, New York (2004)
20. Schneider, K., Brandt, J., Schuele, T., Tuerk, T.: Improving constructiveness in code generators. In: Synchronous Languages, Applications, and Programming (SLAP), Edinburgh, United Kingdom (2005)
21. Schneider, K., Brandt, J., Schuele, T., Tuerk, T.: Maximal causality analysis. In: Application of Concurrency to System Design (ACSD), St. Malo, France, pp. 106–115. IEEE Computer Society, Los Alamitos (2005)
22. Shiple, T.R., Berry, G., Touati, H.: Constructive analysis of cyclic circuits. In: European Design and Test Conference (EDTC), Paris, France. IEEE Computer Society Press, Los Alamitos (1996)