Improving Constructiveness in Code Generators

Klaus Schneider, Jens Brandt, Tobias Schuele, and Thomas Tuerk

University of Kaiserslautern
Department of Computer Science
Reactive Systems Group
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://rsg.informatik.uni-kl.de

Abstract

Perfectly synchronous systems immediately react to the inputs of their environment. These instantaneous reactions may result in so-called causality cycles between the actions of a system and their preconditions. Programs with causality cycles may or may not have consistent and unambiguous behaviors. For this reason, compilers have to perform a causality analysis before code generation. In this paper, we analyze the impact of different code generation schemes on causality analysis and propose translations that yield different degrees of causality. To this end, we first translate the program to an equation system as an intermediate representation, which may alternatively be viewed as a hardware circuit. The second step then analyzes the equation system as known from ternary simulation of hardware circuits with combinational feedback loops. In particular, we consider alternative ways to obtain logically equivalent equation systems that show, however, different results in causality analysis.

Key words: synchronous programming languages, programming language semantics, ternary simulation, causality analysis

1 Introduction

The semantics of synchronous languages [1,17,2,12] is based on the paradigm of *perfect synchrony*, which means that actions are executed as *micro steps* in zero time. Hence, actions immediately change the values of affected variables. As a consequence, *causality cycles* may arise due to cyclic dependencies between preconditions of actions and the modifications that take place due to their execution.

To analyze causality cycles, one has to check whether the considered program has a consistent and unambiguous behavior for all inputs and all reachable states. To this end, typically a formula of type $\forall \boldsymbol{x}.\exists_1 \boldsymbol{y}.\Phi(\boldsymbol{x},\boldsymbol{y})$ has to be proved, which expresses that for all inputs and current states, there must be uniquely determined values for the next states and the current outputs. In the worst case, this requires to

consider the program for all inputs, outputs, and all reachable states, which makes this problem highly complex [1] .

In order to establish more goal-oriented procedures for causality analysis, related problems have been found in many areas of computer science, and the solutions used there have been successfully transferred to causality analysis of synchronous programs. The most interesting relationship is probably the equivalence to the *stabilization of hardware circuits with combinational feedback loops*. The reduction to this problem is natural, since most synchronous programming languages offer simple translations to hardware circuits. As expected, programs with causality cycles then correspond to hardware circuits with combinational feedback loops. Hardware circuits with combinational feedback loops have been considered in detail in [15,14,22,18,19,29,28]. It is well-known that ternary simulation as introduced by Yoeli and Rinon [31] and Eichelberger [11], and further refined by Brzozowski, Bryant, and Seger [7,8,6,9] can be used to analyze the propagation of signal values in these circuits. In particular, the analysis checks whether the signals stabilize for arbitrary gate delays.

Besides ternary simulation, there are many other problems that are equivalent to causality analysis. The following theorem lists some of the most important characterizations and applications (see also [3]):

Theorem 1.1 (Equivalent Characterizations of Causality) *The following problems are all equivalent and can be reduced to each other in polynomial time:*

Causality Analysis of Synchronous Programs [3]:
Given a synchronous program, check without speculative reasoning whether there are unique outputs for all inputs.

Stabilization of Hardware Circuits with Combinational Feedback Loops [27,9]:
Given a combinational hardware circuit, check whether all outputs stabilize for all inputs after some finite time, independently of the delays of the used gates.

Evaluation of Formulas in Intuitionistic Logic [28,21,20,3]:
Given a propositional formula, check whether incomplete truth assignments can be consistently completed by proof rules of intuitionistic logic.

Type Checking [13]:
Check for a given functional program with certain data types whether it is correctly typed.

Existence of Dynamic Schedules [3,10]:
Given a set of mutually dependent guarded actions, check whether there is a dynamic schedule to execute these actions without deadlocks in all possible cases.

Moreover, it is well-known that these problems are co-NP*-complete [18].*

Note, however, that the above problems do not solve the more general problem to check the validity of a (propositional) formula of type $\forall \boldsymbol{x}.\exists_1 \boldsymbol{y}.\Phi(\boldsymbol{x},\boldsymbol{y})$. Instead,

[1] Actually, the precise complexity class is not known. However, the problem is at least in co-NP and at most in PSPACE.

causality analysis may be viewed as a heuristic to solve this problem in the following sense: if causality analysis can prove that there are unique outputs for all inputs, the validity problem has been solved. However, if causality analysis fails, it may still be the case that $\forall \boldsymbol{x}.\exists_1 \boldsymbol{y}.\Phi(\boldsymbol{x},\boldsymbol{y})$ is valid. For this reason, programs with successful causality analysis, which are called *constructive programs*, form a strict subset of programs that have a unique behavior (these are called *logically correct programs* in [3]).

Moreover, causality is a *structural property:* Whether a program is causally correct or not depends on its syntax, and not only on its semantics. In the same way, there are logically equivalent hardware circuits where one has stable signal wires, while the other one may oscillate for some inputs and gate delays. Since causality depends on the syntax of a program, one may ask whether there are reasonable program transformations that turn non-constructive programs to equivalent constructive ones. Another way to achieve this is to directly integrate the mentioned transformations in the code generation.

In this paper, we will answer these questions. To this end, we reconsider Boussinot proposals [5] to improve causality analysis. The problem with these proposals is that they are only given at the level of causality analysis and therefore destroy the equivalences of Theorem 1.1, which is obviously not desired. The main contribution of this paper is to *show how Boussinot's improvements can be integrated in the code generation so that Theorem 1.1 is maintained.* This means that we can modify the intermediate code generation in such a way, that the usual causality analysis of this modified code becomes exactly the causality analysis that Boussinot proposed. The advantage of our approach is clear: we maintain the equivalences of Theorem 1.1, but are nevertheless able to analyze and to compile a broader class of programs. This suggests to have several *degrees of causality,* instead of only declaring a program to be constructive or not. Moreover, in [26], we proved that there is even a *maximal causality analysis*.

The paper has the following outline: in the next section, we consider the basics of equational code generation of Esterel and Quartz programs. To this end, we only consider the computation of the surface actions, since this is sufficient for our purpose. In Section 3, we define the *'can'* and *'must'* approximations by interpreting the data flow in three-valued logic. Section 4 considers these definitions and their consequences in more detail, which is required to explain Boussinot's proposals in Section 5. We also show there how these proposals can be implemented in the code generation so that the usual ternary analysis implements Boussinot's proposals.

2 Computing Actions and Instantaneous Statements

The ideas presented in this paper do not depend on the chosen set of statements. In particular, most proposals consider conditional statements and sequences, which appear in all kinds of imperative programming languages. Nevertheless, to be self-contained, we consider the following set of statements, whose semantics can be found in [4,23,24]:

nothing	(empty statement)
ℓ : **pause**	(separation of macro step)
emit x, **emit next**(x)	(signal emission)
$x := \tau$, **next**$(x) := \tau$	(assignment)
if σ **then** S_1 **else** S_2 **end**	(conditional)
S_1**;** S_2	(sequence)
$S_1 \parallel S_2$ and $S_1 \parallel\!\!\parallel S_2$	(synch./asynch. concurrency)
do S **while** σ	(loop)
[weak] abort S **when [immediate]** σ	(process abortion)
[weak] suspend S **when [immediate]** σ	(process suspension)

Of course, macros like **while** σ **do** S **end** $:\equiv$ **if** σ **then do** S **while** σ **end** can be used to define further statements [4,23,24]. Further basic statements like local declarations are neglected in the following to keep the presentation as readable as possible.

In the following, all basic statements that may change variables are called actions. Hence, actions are of the form **emit** x, **emit next**(x), $x := \tau$, or **next**$(x) := \tau$, where τ is an arbitrary expression. Causality cycles arise due to the immediate effect of actions whose preconditions depend on variable values that are modified by the actions. To explain this in more detail, we have to define the condition when a statement is instantaneously executed.

Definition 2.1 [Instantaneous Execution] *The execution of a statement S is instantaneous iff the predicate* $\mathsf{Inst}\,(S)$ *as defined below is satisfied:*

- $\mathsf{Inst}\,($**nothing**$) := \mathsf{true}$
- $\mathsf{Inst}\,(\ell :$ **pause**$) := \mathsf{false}$
- $\mathsf{Inst}\,(\alpha) := \mathsf{true}$ *for all actions* α
- $\mathsf{Inst}\,($**if** σ **then** S_1 **else** S_2 **end**$) := \sigma \wedge \mathsf{Inst}\,(S_1) \vee \neg\sigma \wedge \mathsf{Inst}\,(S_2)$
- $\mathsf{Inst}\,(S_1; S_2) := \mathsf{Inst}\,(S_1) \wedge \mathsf{Inst}\,(S_2)$
- $\mathsf{Inst}\,(S_1 \parallel S_2) := \mathsf{Inst}\,(S_1) \wedge \mathsf{Inst}\,(S_2)$
- $\mathsf{Inst}\,($**do** S **while** $\sigma$$) := \mathsf{false}$
- $\mathsf{Inst}\,($**while** σ **do** S **end**$) := \neg\sigma$
- $\mathsf{Inst}\,($**[weak] abort** S **when** $\sigma$$) := \mathsf{Inst}\,(S)$
- $\mathsf{Inst}\,($**[weak] abort** S **when immediate** $\sigma$$) := \sigma \vee \mathsf{Inst}\,(S)$
- $\mathsf{Inst}\,($**[weak] suspend** S **when** $\sigma$$) := \mathsf{Inst}\,(S)$
- $\mathsf{Inst}\,(\ell :$ **[weak] suspend** S **when immediate** $\sigma$$) := \neg\sigma \wedge \mathsf{Inst}\,(S)$

The above predicate is used to define the situations where the execution of the considered statement takes no time. This is required to define the actions that a statement can execute at the current point of time as shown below. Of course, to compile a program, one has to compute *all* actions of the program, but the improvements proposed in this paper can already be described with the 'surface' actions.

Definition 2.2 [Surface Actions] *For a given statement S and a precondition φ, the following recursive definition can be used to compute a set of guarded actions (γ, α) such that action α is immediately executed whenever γ currently holds:*

- $\mathsf{ActSf}\,(\varphi, \textbf{nothing}) := \{\}$
- $\mathsf{ActSf}\,(\varphi, \ell : \textbf{pause}) := \{\}$
- $\mathsf{ActSf}\,(\varphi, \alpha) := \{(\varphi, \alpha)\}$ *for actions* α
- $\mathsf{ActSf}\,(\varphi, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) := \mathsf{ActSf}\,(\varphi \wedge \sigma, S_1) \cup \mathsf{ActSf}\,(\varphi \wedge \neg\sigma, S_2)$
- $\mathsf{ActSf}\,(\varphi, S_1; S_2) := \mathsf{ActSf}\,(\varphi, S_1) \cup \mathsf{ActSf}\,(\varphi \wedge \mathsf{Inst}\,(S_1), S_2)$
- $\mathsf{ActSf}\,(\varphi, S_1 \parallel S_2) := \mathsf{ActSf}\,(\varphi, S_1) \cup \mathsf{ActSf}\,(\varphi, S_2)$
- $\mathsf{ActSf}\,(\varphi, \textbf{do } S \textbf{ while } \sigma) := \mathsf{ActSf}\,(\varphi, S)$
- $\mathsf{ActSf}\,(\varphi, \textbf{while } \sigma \textbf{ do } S \textbf{ end}) := \mathsf{ActSf}\,(\varphi \wedge \sigma, S)$
- $\mathsf{ActSf}\,(\varphi, \textbf{[weak] abort } S \textbf{ when } \sigma) := \mathsf{ActSf}\,(\varphi, S)$
- $\mathsf{ActSf}\,(\varphi, \textbf{weak abort } S \textbf{ when immediate } \sigma) := \mathsf{ActSf}\,(\varphi, S)$
- $\mathsf{ActSf}\,(\varphi, \textbf{abort } S \textbf{ when immediate } \sigma) := \mathsf{ActSf}\,(\varphi \wedge \neg\sigma, S)$
- $\mathsf{ActSf}\,(\varphi, \textbf{[weak] suspend } S \textbf{ when } \sigma) := \mathsf{ActSf}\,(\varphi, S)$
- $\mathsf{ActSf}\,(\varphi, \ell : \textbf{weak suspend } S \textbf{ when immediate } \sigma) := \mathsf{ActSf}\,(\varphi, S)$
- $\mathsf{ActSf}\,(\varphi, \ell : \textbf{suspend } S \textbf{ when immediate } \sigma) := \mathsf{ActSf}\,(\varphi \wedge \neg\sigma, S)$

module P_2 :
output o_1, o_2;
$S_2 :=$ **emit** o_2;
if o_1 **then**
if o_2 **then**
ℓ : **pause**
end;
emit o_1
end
end module

Figure 1. Program P_2

As an example, for the statement S_2 given in Figure 1, we obtain $\mathsf{Inst}\,(S_2) = \neg o_1 \vee \neg o_2$ and $\mathsf{ActSf}\,(\mathsf{true}, S_2) = \{(\mathsf{true}, \textbf{emit } o_2), (o_1 \wedge \neg o_2, \textbf{emit } o_1)\}$. As can be seen, the precondition γ of a guarded action (γ, α) may depend on values that are changed by the corresponding action α. For example, precondition $o_1 \wedge \neg o_2$ depends on o_1, which is set by the corresponding action **emit** o_1. Such dependencies are called *causality cycles*.

3 Causality Analysis Based on Three-Valued Logic

Causality analysis can be explained in several ways, since the problem and its related algorithms are fundamental for many areas of computer science. In the following, we use three-valued logic to describe the problem and ternary simulation

as a method to solve it. To this end, we extend the Boolean values $\mathsf{false}, \mathsf{true}$ by a third truth value $\bot$ and define a partial order on these three values as follows:

$$x \sqsubseteq y :\Leftrightarrow x = \bot \vee x = y$$

It is easily seen that the set $\{\bot, \mathsf{false}, \mathsf{true}\}$ ordered by $\sqsubseteq$ is a (semi-)lattice, since we have a greatest lower bound for all $x, y \in \{\bot, \mathsf{false}, \mathsf{true}\}$. The same holds for the product space $\{\bot, \mathsf{false}, \mathsf{true}\}^n$, when $\sqsubseteq^n$ is defined as the componentwise application of $\sqsubseteq$. In the following, we have to consider environments $\mathcal{E}$, which are functions that map (input, output, and location) variables to values of $\{\bot, \mathsf{false}, \mathsf{true}\}$. We also establish a partial order on these environments as follows:

$$\mathcal{E}_1 \preceq \mathcal{E}_2 :\Leftrightarrow \text{ for all variables } x,\ \mathcal{E}_1(x) \neq \bot \text{ implies } \mathcal{E}_1(x) = \mathcal{E}_2(x).$$

Boolean functions can be represented as propositional logic formulas. Moreover, it is well-known that the operators $\neg$, $\wedge$, $\vee$ are sufficient for that purpose. Hence, it is sufficient to extend the basic Boolean functions represented by $\neg$, $\wedge$, $\vee$ to the ternary domain. Other Boolean functions are then simply obtained by composition of these basic functions (see [25,26] for a discussion of these definitions).

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

x	$\neg x$
$\bot$	$\bot$
0	1
1	0

Figure 2. Ternary extension of basic Boolean functions

Ternary extensions of the basic Boolean operations are given in Figure 2. As can be seen, these functions are monotonic for the above mentioned partial order $\sqsubseteq$. For this reason, the Tarski-Knaster theorem can be applied [30,16,25], which guarantees the existence of least fixpoints of monotonic functions. Moreover, the least fixpoint can be computed by the Tarski-Knaster fixpoint iteration.

To do so, we must identify the function whose fixpoint is to be computed. To this end, we give the following definition of pessimistic and optimistic estimations of actions of the surface:

Definition 3.1 [Estimating Executed Actions]*Given a statement S, the set of actions of its surface that can and must be executed under a precondition φ, respectively, are defined as follows:*

$$\begin{aligned} \mathsf{CanAct}\,(\mathcal{E}, S) &:= \{\alpha \mid \exists \gamma.(\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S) \wedge \mathcal{E}(\gamma) \neq \mathsf{false}\} \\ \mathsf{MustAct}\,(\mathcal{E}, S) &:= \{\alpha \mid \exists \gamma.(\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S) \wedge \mathcal{E}(\gamma) = \mathsf{true}\} \end{aligned}$$

Note that $\mathsf{CanAct}\,(\mathcal{E}, S) = \mathsf{MustAct}\,(\mathcal{E}, S)$ holds iff all guards γ of the surface actions $(\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S)$ have Boolean values $\mathcal{E}(\gamma)$ under the current environment $\mathcal{E}$, i.e., if the guards have been determined. By the above definition, we moreover have the following consequences:

Lemma 3.2 (Properties of Can and Must) *For every statement S and every environment $\mathcal{E}$, we have* $\mathsf{MustAct}(\mathcal{E}, S) \subseteq \mathsf{CanAct}(\mathcal{E}, S)$. *Moreover, for environments $\mathcal{E}_1$ and $\mathcal{E}_2$ with $\mathcal{E}_1 \preceq \mathcal{E}_2$, we have*

- $\mathcal{E}_1(\mathsf{Inst}(S)) \sqsubseteq \mathcal{E}_2(\mathsf{Inst}(S))$
- $\mathsf{CanAct}(\mathcal{E}_1, S) \subseteq \mathsf{CanAct}(\mathcal{E}_2, S)$
- $\mathsf{MustAct}(\mathcal{E}_1, S) \subseteq \mathsf{MustAct}(\mathcal{E}_2, S)$

Causality analysis starts with the three-valued environment $\mathcal{E}_0$, where $\mathcal{E}_0(x) = \bot$ holds for all output variables x, and where $\mathcal{E}_0(x) \in \{\mathsf{true}, \mathsf{false}\}$ holds for all other variables x. In the following, the environment is updated so that a monotonic sequence $\mathcal{E}_0 \preceq \mathcal{E}_1 \preceq \mathcal{E}_2 \ldots$ is obtained. This update is defined as follows: given environment $\mathcal{E}_i$, we compute $\mathcal{D}_{\mathsf{can}} := \mathsf{CanAct}(\mathcal{E}_i, S)$ and $\mathcal{D}_{\mathsf{must}} := \mathsf{MustAct}(\mathcal{E}_i, S)$. According to the above lemma, one of the following cases must hold for every action **emit** x (we neglect delayed emissions [25] and assignments here):

- if **emit** $x \in \mathcal{D}_{\mathsf{must}}$ holds, then we change the value of x to true
- if **emit** $x \notin \mathcal{D}_{\mathsf{can}}$ holds, then we change the value of x to false
- if **emit** $x \in \mathcal{D}_{\mathsf{can}} \setminus \mathcal{D}_{\mathsf{must}}$ holds, then we cannot change the value of x

It can be easily seen that the environments $\mathcal{E}_i$ obtained by the above iteration form a monotonic sequence, so that after finitely many steps a fixpoint $\check{\mathcal{E}}$ is reached. If the values of all output variables have been determined, then the reaction of the program has been constructively determined.

function $ComputeOutputs(\mathcal{E}_0, S, \mathcal{V}_{\mathsf{out}})$
  $\mathcal{E} = \mathcal{E}_0$;
  do
    $\mathcal{E}_{\mathsf{old}} := \mathcal{E}$;
    $\mathcal{D}_{\mathsf{can}} := \mathsf{CanAct}(\mathcal{E}, S)$;
    $\mathcal{D}_{\mathsf{must}} := \mathsf{MustAct}(\mathcal{E}, S)$;
    for all $x \in \mathcal{V}_{\mathsf{out}}$ **do**
      if emit $x \in \mathcal{D}_{\mathsf{must}}$ **then** $\mathcal{E} := \mathcal{E}_x^{\mathsf{true}}$ **end**;
      if emit $x \notin \mathcal{D}_{\mathsf{can}}$ **then** $\mathcal{E} := \mathcal{E}_x^{\mathsf{false}}$ **end**
    end
  while $\mathcal{E}_{\mathsf{old}} \neq \mathcal{E}$
  return $(\mathcal{D}_{\mathsf{can}}, \mathcal{E})$
end

Figure 3. Causality analysis

Figure 3 shows the entire algorithm that computes the outputs and the actions for given inputs (encoded in $\mathcal{E}_0$). The argument $\mathcal{E}_0$ is thereby the initial environment, S is the considered statement, and $\mathcal{V}_{\mathsf{out}}$ is the set of declared output signals. $\mathcal{E}_x^v$ means that the value of $\mathcal{E}(x)$ is changed to v, while all other values are maintained. Note that the computations of $\mathcal{D}_{\mathsf{can}}$ and $\mathcal{D}_{\mathsf{must}}$ can be based on a precomputation of $\mathsf{ActSf}(\varphi, S)$ so that only the guards have to be re-evaluated. Finally, the procedure

has to be repeated for all reachable states and must, moreover, be generalized to delayed actions [28,25].

4 Problematic Cases for Causality

The algorithm described in the previous section is the essence of standard procedures for causality analysis. In the next section, we will discuss several proposals to refine the definition of the surface actions so that the standard causality analysis becomes more powerful. To this end, we first consider in this section the consequences of the previous definition of $\mathsf{CanAct}\,(\mathcal{E}, S)$ and $\mathsf{MustAct}\,(\mathcal{E}, S)$.

Lemma 4.1 (Recursive Definition of $\mathsf{CanAct}\,(\mathcal{E}, S)$) *For every statement S and every environment $\mathcal{E}$, the following propositions hold:*

- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{nothing}) = \{\}$
- $\mathsf{CanAct}\,(\mathcal{E}, \ell : \textbf{pause}) = \{\}$
- $\mathsf{CanAct}\,(\mathcal{E}, \alpha) = \{\alpha\}$ *for all actions* α
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$
$$= \begin{cases} \mathsf{CanAct}\,(\mathcal{E}, S_1) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{true} \\ \mathsf{CanAct}\,(\mathcal{E}, S_2) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{false} \\ \mathsf{CanAct}\,(\mathcal{E}, S_1) \cup \mathsf{CanAct}\,(\mathcal{E}, S_2) & : \mathit{if}\ \mathcal{E}(\sigma) = \bot \end{cases}$$
- $\mathsf{CanAct}\,(\mathcal{E}, S_1; S_2)$
$$= \begin{cases} \mathsf{CanAct}\,(\mathcal{E}, S_1) & : \mathit{if}\ \mathcal{E}(\mathsf{Inst}\,(S_1)) = \mathsf{false} \\ \mathsf{CanAct}\,(\mathcal{E}, S_1) \cup \mathsf{CanAct}\,(\mathcal{E}, S_2) & : \mathit{if}\ \mathcal{E}(\mathsf{Inst}\,(S_1)) \in \{\bot, \mathsf{true}\} \end{cases}$$
- $\mathsf{CanAct}\,(\mathcal{E}, S_1 \parallel S_2) = \mathsf{CanAct}\,(\mathcal{E}, S_1) \cup \mathsf{CanAct}\,(\mathcal{E}, S_2)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{do } S \textbf{ while } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{while } \sigma \textbf{ do } S \textbf{ end}) = \begin{cases} \{\} & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{false} \\ \mathsf{CanAct}\,(\mathcal{E}, S) & : \mathit{if}\ \mathcal{E}(\sigma) \in \{\bot, \mathsf{true}\} \end{cases}$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{abort } S \textbf{ when } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{[weak] abort } S \textbf{ when } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{weak abort } S \textbf{ when immediate } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{abort } S \textbf{ when immediate } \sigma) = \begin{cases} \{\} & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{true} \\ \mathsf{CanAct}\,(\mathcal{E}, S) & : \mathit{otherwise} \end{cases}$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{suspend } S \textbf{ when } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{[weak] suspend } S \textbf{ when } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{weak suspend } S \textbf{ when immediate } \sigma) = \mathsf{CanAct}\,(\mathcal{E}, S)$
- $\mathsf{CanAct}\,(\mathcal{E}, \textbf{suspend } S \textbf{ when immediate } \sigma)$
$$= \begin{cases} \{\} & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{true} \\ \mathsf{CanAct}\,(\mathcal{E}, S) & : \mathit{otherwise} \end{cases}$$

The above lemma shows that the recursion of $\mathsf{CanAct}\,(\mathcal{E}, S)$ is straightforward, which is the preferred definition in [3]. The same holds for $\mathsf{MustAct}\,(\mathcal{E}, S)$ with the exception of conditionals as will be discussed in the next section:

Lemma 4.2 (Recursive Definition of $\mathsf{MustAct}\,(\mathcal{E}, S)$) *For every statement S and every environment $\mathcal{E}$, the following propositions hold:*

- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{nothing}) = \{\}$
- $\mathsf{MustAct}\,(\mathcal{E}, \ell : \textbf{pause}) = \{\}$
- $\mathsf{MustAct}\,(\mathcal{E}, \alpha) = \{\alpha\}$ *for all actions* α
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S_1) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{true} \\ \mathsf{MustAct}\,(\mathcal{E}, S_2) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{false} \\ \{\} & : \mathit{if}\ \mathcal{E}(\sigma) = \bot \end{cases}$
- $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$
  $= \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S_1) \cup \mathsf{MustAct}\,(\mathcal{E}, S_2) & : \mathit{if}\ \mathcal{E}(\mathsf{Inst}\,(S_1)) = \mathsf{true} \\ \mathsf{MustAct}\,(\mathcal{E}, S_1) & : \mathit{if}\ \mathcal{E}(\mathsf{Inst}\,(S_1)) \in \{\bot, \mathsf{false}\} \end{cases}$
- $\mathsf{MustAct}\,(\mathcal{E}, S_1 \parallel S_2) = \mathsf{MustAct}\,(\mathcal{E}, S_1) \cup \mathsf{MustAct}\,(\mathcal{E}, S_2)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{do } S \textbf{ while } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{while } \sigma \textbf{ do } S \textbf{ end}) = \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{true} \\ \{\} & : \mathit{otherwise} \end{cases}$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{abort } S \textbf{ when } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{[weak] abort } S \textbf{ when } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{weak abort } S \textbf{ when immediate } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{abort } S \textbf{ when immediate } \sigma)$
  $= \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{false} \\ \{\} & : \mathit{otherwise} \end{cases}$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{suspend } S \textbf{ when } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{[weak] suspend } S \textbf{ when } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{weak suspend } S \textbf{ when immediate } \sigma) = \mathsf{MustAct}\,(\mathcal{E}, S)$
- $\mathsf{MustAct}\,(\mathcal{E}, \textbf{suspend } S \textbf{ when immediate } \sigma)$
  $= \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S) & : \mathit{if}\ \mathcal{E}(\sigma) = \mathsf{false} \\ \{\} & : \mathit{otherwise} \end{cases}$

The above recursive characterizations of $\mathsf{CanAct}\,(\mathcal{E}, S)$ and $\mathsf{MustAct}\,(\mathcal{E}, S)$ are direct consequences of the three-valued version of $\mathsf{ActSf}\,(\varphi, S)$. Ignoring this origin, one may think of the following alternative definition of $\mathsf{MustAct}\,(\mathcal{E}, S)$ (with the abbreviations $M_1 := \mathsf{MustAct}\,(\mathcal{E}, S_1)$ and $M_2 := \mathsf{MustAct}\,(\mathcal{E}, S_2)$):

$$\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} M_1 & : \text{if } \mathcal{E}(\sigma) = \mathsf{true} \\ M_2 & : \text{if } \mathcal{E}(\sigma) = \mathsf{false} \\ M_1 \cap M_2 & : \text{if } \mathcal{E}(\sigma) = \bot \end{cases}$$

This has already been discussed by Berry in [3] (page 81). He rejects this modification, since it 'performs speculative reasoning'. In his view, this modification would allow 'backward information flow'. However, in our opinion, this view is a bit inconsistent, since the definition of $\mathsf{CanAct}\,(\ldots)$ also allows a backward information flow for conditionals. Hence, although we agree that the modification can not be simply used with the normal code generation, we think that the reason to reject it is simply the incompatibility with code generation. In this paper, we show how code generation can be made compatible so that the modification can be perfectly used in compilers.

Boussinot proposed further alternatives for the definition of $\mathsf{CanAct}\,(\mathcal{E}, S)$ and $\mathsf{MustAct}\,(\mathcal{E}, S)$ [5]. *Again, just changing the above estimations for causality analysis without corresponding changes of the code generation is not acceptable, since Theorem 1.1 would no longer hold!* Thus, the compilers would succeed with the causality analysis, but the generated hardware circuits would possibly not stabilize or the generated software code would suffer from deadlocks. Using the above modification of the must-approximation of the conditional would allow us, for example, to prove the causality of program P_{12} given in Figure 5. However, a circuit derived from that program which is based on the definition of $\mathsf{ActSf}\,(\mathsf{true}, \varphi)$ would not stabilize.

Hence, our aim is not to directly change $\mathsf{CanAct}\,(\mathcal{E}, S)$ and $\mathsf{MustAct}\,(\mathcal{E}, S)$ to improve causality analysis. Instead, our aim is to retain Definition 3.1, and instead to improve the definition of the surface actions, so that Theorem 1.1 still holds. Of course, this also changes the sets that were obtained by computing $\mathsf{CanAct}\,(\mathcal{E}, S)$ and $\mathsf{MustAct}\,(\mathcal{E}, S)$. However, this is *consistent with the generated code.*

5 Implementing Boussinot's Improvements

We now show how the circuit code generation, in particular, the definition of $\mathsf{ActSf}\,(\varphi, S)$ can be changed such that the standard causality analysis is able to handle more cyclic programs. To this end, we show that Boussinot's improvements [5] can be implemented by appropriate changes in the circuit code generator and by the standard causality analysis.

5.1 *Proposal 1: Attempt to Refine* $\mathsf{CanInstant}\,(\mathcal{E}, S)$

Recall that the definition of $\mathsf{ActSf}\,(\varphi, S)$ depends on the definition of $\mathsf{Inst}\,(S)$. To be precise, the definition of the surface actions of sequences depends on $\mathsf{Inst}\,(S)$:

$$\mathsf{CanAct}\,(\mathcal{E}, S_1; S_2) = \begin{cases} \mathsf{CanAct}\,(\mathcal{E}, S_1) & : \text{if } \mathcal{E}(\mathsf{Inst}\,(S_1)) = \mathsf{false} \\ \mathsf{CanAct}\,(\mathcal{E}, S_1) \cup \mathsf{CanAct}\,(\mathcal{E}, S_2) & : \text{otherwise} \end{cases}$$

One might think of introducing a new predicate $\mathsf{CanInstant}\,(\mathcal{E}, S)$ which is used instead of $\mathcal{E}(\mathsf{Inst}\,(S_1))$ such that $\mathsf{CanAct}\,(\mathcal{E}, S_1; S_2)$ becomes a smaller set. Hence,

$\mathsf{CanInstant}\,(\mathcal{E}, S_1)$ should be more often false than $\mathcal{E}(\mathsf{Inst}\,(S_1))$. To this end, we may try the following definition:

$$\mathsf{CanInstant}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} inst_1 & : \text{if } \mathcal{E}(\sigma) = \mathsf{true} \\ inst_2 & : \text{if } \mathcal{E}(\sigma) = \mathsf{false} \\ f_i(inst_1, inst_2) & : \text{if } \mathcal{E}(\sigma) = \bot \end{cases}$$

with $inst_i := \mathsf{CanInstant}\,(\mathcal{E}, S_i)$ and one of the following functions f_i:

- $f_0(inst_1, inst_2) := \bot$
- $f_1(inst_1, inst_2) := \bot \wedge inst_1 \vee \neg\bot \wedge inst_2$
- $f_2(inst_1, inst_2) := \bot \wedge inst_1 \vee \neg\bot \wedge inst_2 \vee inst_1 \wedge inst_2$
- $f_3(inst_1, inst_2) := inst_1 \vee inst_2$

It can be proved that $\mathcal{E}(\mathsf{Inst}\,(S)) = \mathsf{CanInstant}\,(\mathcal{E}, S)$ holds if we use function f_1. However, as $\bot$ and true are not distinguished in the definition of $\mathsf{CanAct}\,(\mathcal{E}, S_1; S_2)$ above, the versions of $\mathsf{CanInstant}\,(\mathcal{E}, S)$ with f_1, f_2, and f_3 are all equivalent, which can be easily seen by inspecting the truth tables below. Thus, this proposal does not lead to improvements for causality analysis.

f_0	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	$\bot$	$\bot$
1	$\bot$	$\bot$	$\bot$

f_1	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$
1	$\bot$	$\bot$	$\bot$

f_2	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$
1	$\bot$	$\bot$	1

f_3	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

5.2 *Proposal 2: Attempt to Refine* $\mathsf{MustInstant}\,(\mathcal{E}, S)$

In a similar way as in the previous section, we could try to improve the definition of $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$. Recall that the current definition leads to the following recursion:

$$\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2) = \begin{cases} \mathsf{MustAct}\,(\mathcal{E}, S_1) \\ \quad \cup \mathsf{MustAct}\,(\mathcal{E}, S_2) & : \text{if } \mathcal{E}(\mathsf{Inst}\,(S_1)) = \mathsf{true} \\ \mathsf{MustAct}\,(\mathcal{E}, S_1) & : \text{otherwise} \end{cases}$$

With the same arguments as in the previous section, one might think of a new predicate $\mathsf{MustInstant}\,(\mathcal{E}, S)$ such that $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$ becomes a larger set. Hence, $\mathsf{MustInstant}\,(\mathcal{E}, S)$ should be more often true than $\mathcal{E}(\mathsf{Inst}\,(S_1))$. Again, the only interesting case where the definition can be changed is for conditionals, and similar as in the previous section, we consider now the following definition with $inst_i := \mathsf{MustInstant}\,(\mathcal{E}, S_i)$:

$$\mathsf{MustInstant}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} inst_1 & : \text{if } \mathcal{E}(\sigma) = \mathsf{true} \\ inst_2 & : \text{if } \mathcal{E}(\sigma) = \mathsf{false} \\ g_i(inst_1, inst_2) & : \text{if } \mathcal{E}(\sigma) = \bot \end{cases}$$

with one of the following functions g_i:

module P_{10} :
  output o;
    if o **then nothing end**;
    emit o
end module

Figure 4. Program P_{10}

- $g_0(inst_1, inst_2) := \bot$
- $g_1(inst_1, inst_2) := \bot \wedge inst_1 \vee \neg\bot \wedge inst_2$
- $g_2(inst_1, inst_2) := \bot \wedge inst_1 \vee \neg\bot \wedge inst_2 \vee inst_1 \wedge inst_2$
- $g_3(inst_1, inst_2) := inst_1 \wedge inst_2$

Considering the definition of $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$, it is clear that improvements of $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$ are only found when some functions g_i differ between $\bot$ and true. Hence, g_0 and g_1 lead to the same definition of $\mathsf{MustAct}\,(\mathcal{E}, S_1; S_2)$, and for the same reason, also g_2 and g_3 have the same effect, which can be seen by the truth tables below. However, there are relevant differences between g_1 and g_2.

g_0	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	$\bot$	$\bot$
1	$\bot$	$\bot$	$\bot$

g_1	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$
1	$\bot$	$\bot$	$\bot$

g_2	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$
1	$\bot$	$\bot$	1

g_3	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

The problem is now that we have to change the definition of $\mathsf{Inst}\,(S)$ to mimic the above effects. It is easily seen that all we have to do is the following change:

$$\mathsf{Inst}\,(\textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) := \begin{pmatrix} \sigma \wedge \mathsf{Inst}\,(S_1) \vee \\ \neg\sigma \wedge \mathsf{Inst}\,(S_2) \vee \\ \mathsf{Inst}\,(S_1) \wedge \mathsf{Inst}\,(S_2) \end{pmatrix}$$

Note that using the above modification of $\mathsf{Inst}\,(S)$ will also generate software and hardware code such that Theorem 1.1 remains valid. However, more programs can now be compiled: For example, program P_{10} given in Figure 4 will become now constructive. Using the original definition, we would obtain the surface actions $\{(o \vee \neg o, \textbf{emit } o)\}$ which lead to the equation (hardware circuit) $o = o \vee \neg o$ which turns out not to be constructive (since in intuitionistic logic $o \vee \neg o$ can not be proved). Using the modified definition, we obtain the surface actions $\{(o \vee \neg o \vee \mathsf{true}, \textbf{emit } o)\}$, i.e., $\{(\mathsf{true}, \textbf{emit } o)\}$ which leads to the acyclic equation (hardware circuit) $o = \mathsf{true}$.

5.3 Proposal 3: Refine $\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$

The next proposal modifies the definition of $\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$. If the same actions appear in both $\mathsf{MustAct}\,(\mathcal{E}, S_1)$ and $\mathsf{MustAct}\,(\mathcal{E}, S_2)$, it is clear

module P_{12} :
output o;
  if o **then emit** o
  else emit o
  end
end module

Figure 5. Program P_{12}

that these have to be executed, regardless what the value of $\mathcal{E}(\sigma)$ is. Hence, we would like to use the following definition (with $M_i := \mathsf{MustAct}\,(\mathcal{E}, S_i)$):

$$\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} M_1 & : \text{if } \mathcal{E}(\sigma) = \mathsf{true} \\ M_2 & : \text{if } \mathcal{E}(\sigma) = \mathsf{false} \\ M_1 \cap M_2 & : \text{if } \mathcal{E}(\sigma) = \bot \end{cases}$$

We already discussed this modification at the end of the previous section. Obviously, $\mathsf{MustAct}\,(\mathcal{E}, S)$ becomes a larger set, and therefore, this leads to a more powerful definition of causality. Again, we have to change the code generation correspondingly, i.e., the definition of $\mathsf{ActSf}\,(\varphi, S)$, such that together with Definition 3.1, we obtain the above recursive definition. To this end, we propose the following changes for the definition of $\mathsf{ActSf}\,(\varphi, S)$ with $A_i := \mathsf{ActSf}\,(\varphi, S_i)$:

$$\begin{aligned} &\mathsf{ActSf}\,(\varphi, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) \\ &:= \begin{pmatrix} \{(\gamma_1 \wedge \sigma, \alpha) \mid (\gamma_1, \alpha) \in A_1\} \cup \\ \{(\gamma_2 \wedge \neg\sigma, \alpha) \mid (\gamma_2, \alpha) \in A_2\} \cup \\ \{(\gamma_1 \wedge \gamma_2, \alpha) \mid (\gamma_1, \alpha) \in A_1 \wedge (\gamma_2, \alpha) \in A_2\} \end{pmatrix} \end{aligned}$$

The idea is as follows: if an action α appears in both A_1 and A_2 with guards γ_1 and γ_2, then the condition $\gamma_1 \wedge \sigma \vee \gamma_2 \wedge \neg\sigma$ is responsible for the execution of α. However, if σ can not be determined, then causality analysis may fail. It is easily seen that $\gamma_1 \wedge \sigma \vee \gamma_2 \wedge \neg\sigma$ is equivalent to $\gamma_1 \wedge \sigma \vee \gamma_2 \wedge \neg\sigma \vee \gamma_1 \wedge \gamma_2$, and therefore we are allowed to add these guarded actions. Even if σ can not be determined, causality analysis may now succeed if $\gamma_1 \wedge \gamma_2$ holds.

Again, this modification of $\mathsf{ActSf}\,(\varphi, S)$ will retain Theorem 1.1. However, more programs can now be compiled: For example, program P_{12} given in Figure 5 becomes constructive: Using the original definition, we would obtain the surface actions $\{(o, \textbf{emit } o), (\neg o, \textbf{emit } o)\}$ which lead to the equation (hardware circuit) $o = o \vee \neg o$ which is not constructive. Using the modified definition, we obtain (with propagation of Boolean constants) the surface actions $\{(\mathsf{true}, \textbf{emit } o)\}$ which leads to the acyclic equation (hardware circuit) $o = \mathsf{true}$.

module P_{16} :
output o;
  if o **then**
    if o **else emit** o **end**
  end
end

Figure 6. Program P_{16}

5.4 *Proposal 4: Refine* $\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$

Boussinot also proposed to update the environment $\mathcal{E}$ whenever a program part is analyzed which requires that some particular values of the environment must hold. In particular, he proposed to modify $\mathsf{MustAct}\,(\mathcal{E}, S)$ as follows:

$$\mathsf{MustAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \begin{cases} \mathsf{MustAct}\,(\mathcal{E}_\sigma^{\mathsf{true}}, S_1) & : \text{if } \mathcal{E}(\sigma) = \mathsf{true} \\ \mathsf{MustAct}\,(\mathcal{E}_\sigma^{\mathsf{false}}, S_2) & : \text{if } \mathcal{E}(\sigma) = \mathsf{false} \\ \mathsf{MustAct}\,(\mathcal{E}, S_1) \cap \mathsf{MustAct}\,(\mathcal{E}, S_2) & : \text{if } \mathcal{E}(\sigma) = \bot \end{cases}$$

and similarly for $\mathsf{CanAct}\,(\mathcal{E}, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$. Also these modifications can be brought to the code generation level with the following changed definition (with $A_i := \mathsf{ActSf}\,(\varphi, S_i)$), where $[\gamma]_\sigma^\tau$ means to replace every occurrence of σ in γ with τ:

$$\mathsf{ActSf}\,(\varphi, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) := \begin{pmatrix} \{([\gamma_1]_\sigma^{\mathsf{true}} \wedge \sigma, \alpha) \mid (\gamma_1, \alpha) \in A_1\} \cup \\ \{([\gamma_2]_\sigma^{\mathsf{false}} \wedge \neg\sigma, \alpha) \mid (\gamma_2, \alpha) \in A_2\} \cup \\ \{(\gamma_1 \wedge \gamma_2, \alpha) \mid (\gamma_1, \alpha) \in A_1 \wedge (\gamma_2, \alpha) \in A_2\} \end{pmatrix}$$

In a similar way, we can also use updated environments for other statements, in particular for abortion and suspension:

- $\mathsf{ActSf}\,(\varphi, \textbf{abort } S \textbf{ when immediate } \sigma)$
  $:= \{([\gamma]_\sigma^{\mathsf{false}} \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S)\}$
- $\mathsf{ActSf}\,(\varphi, \textbf{suspend } S \textbf{ when immediate } \sigma)$
  $:= \{([\gamma]_\sigma^{\mathsf{false}} \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S)\}$

Again, the above modification of $\mathsf{ActSf}\,(\varphi, S)$ will retain Theorem 1.1. However, more programs can now be compiled: For example, program P_{16} given in Figure 6 will now become constructive. Using the original definition, we obtain the surface action $\{(o \wedge \neg o, \textbf{emit } o)\}$ which leads to the non-constructive equation $o = o \wedge \neg o$. Using the modified definition, we obtain the surface action $\{([\neg o]_o^{\mathsf{true}}, \textbf{emit } o)\}$, i.e., $\{(\mathsf{false}, \textbf{emit } o)\}$ which leads to the acyclic equation $o = \mathsf{false}$.

module P_{17} :
output o_1, o_2;
  if o_1 **then**
    emit o_2;
    if o_2 **else emit** o_1 **end**
  end
end

Figure 7. Program P_{17}

5.5 *Proposal 5: Refine* $\mathsf{MustAct}\,(\mathcal{E}, \textbf{emit}\ o; S)$

The last proposal we consider updates the environment if an emission is passed in a sequence. Hence, we modify the definition of $\mathsf{ActSf}\,(\varphi, S_1; S_2)$ in case S_1 is an emission as follows:

$$\mathsf{ActSf}\,(\varphi, \textbf{emit}\ o; S) = \{(\varphi, \textbf{emit}\ o)\} \cup \{([\gamma]_o^{\mathsf{true}}, \alpha) \mid (\gamma, \alpha) \in \mathsf{ActSf}\,(\varphi, S)\}$$

Again, some programs become constructive with the above definition that were not constructive with the original definition. As an example, consider program P_{17} in Figure 7. Using the original definition, we would obtain the actions $\{(o_1 \wedge \neg o_2, \textbf{emit}\ o_1), (o_1, \textbf{emit}\ o_2)\}$ which lead to the following equation system (hardware circuit): $\{o_1 = o_1 \wedge \neg o_2, o_2 = o_1\}$. It is easily seen that this equation system can not be solved by ternary simulation, since it contains again the problem to prove or disprove $x \wedge \neg x$ which is not valid in ternary logic. Using the modified definition, we obtain instead the surface actions $\{(o_1 \wedge [\neg o_2]_{o_2}^{\mathsf{true}}, \textbf{emit}\ o_1), (o_1, \textbf{emit}\ o_2)\}$, i.e., $\{(\mathsf{false}, \textbf{emit}\ o_1), (o_1, \textbf{emit}\ o_2)\}$, and therefore the simple acyclic equation system $\{o_1 = \mathsf{false}, o_2 = o_1\}$.

5.6 *Further Improvements*

In the previous five subsections, we have reviewed Boussinot's proposals for improving causality analysis. This imposes the question if there are further improvements, in particular, modifications of the code generation, i.e., the recursive definition of $\mathsf{ActSf}\,(\varphi, S)$.

First of all, we can clearly demonstrate that further modifications to improve causality analysis are possible. In particular, we have proved in [26] that once the equation system is obtained, adding all prime implicants yields an equivalent equation system whose constructiveness implies the constructiveness of the original one. Moreover, we proved in [26] that this yields a *maximal causality* analysis.

```
module P18 :
output o1, o2;
  if o1 then
       emit o2
    ||
       if o2 else emit o1 end
  end
end
```

```
module P19 :
output o1, o2, o3, o4;
  if o2 then emit o1 end
||
  if o1 ∧ o3 then emit o2 end
||
  if ¬o1 ∧ o4 then emit o2 end
||
  emit o3
||
  emit o4
end
```

Figure 8. Programs P_{18} and P_{19}

As an example, consider program P_{19} given in Figure 8: we obtain the following equation system E_{19} for the starting time of the program:

$$E_{19} := \begin{cases} o_1 = o_2 \\ o_2 = o_1 \wedge o_3 \vee \neg o_1 \wedge o_4 \\ o_3 = \mathsf{true} \\ o_4 = \mathsf{true} \end{cases}$$

The above set of equations is not constructive, and none of the proposals in this paper can be used to generate a better equation system. However, adding the missing prime implicant $o_3 \wedge o_4$ in the second equation yields the equation

$$o_2 = o_1 \wedge o_3 \vee \neg o_1 \wedge o_4 \vee o_3 \wedge o_4,$$

which makes the code constructive. Hence, adding prime implicants is more powerful than the proposals of the previous sections. However, it is also more complex.

Nevertheless, also *maximal causality analysis is not able to replace logical correctness:* Program P_{18} given in Figure 8 yields $\{(o_1 \wedge \neg o_2, \textbf{emit } o_1), (o_1, \textbf{emit } o_2)\}$ with the original code generator, and also with the maximal causality analysis. Although program P_{18} has a unique behavior, this unique behavior can not even be found by maximal causality analysis.

6 Conclusions

In this paper, we reviewed Boussinot's proposals [5] for improving causality analysis. In contrast to Boussinot's work, we did not simply modify the definitions of $\mathsf{CanAct}(\mathcal{E}, S)$ and $\mathsf{MustAct}(\mathcal{E}, S)$. Instead, we changed the basis for code generation, i.e., the computation of guarded commands for the data flow such that Boussinot's proposals were obtained via the ternary interpretation of the data flow. This is necessary to retain the beautiful relationship between causality analysis and characterizations of the generated code like existence of dynamic schedules and

module P_{12a} :
output o_1, o_2;
  if o_1 **then emit** o_2
  else emit o_2
  end
end module

Figure 9. Program P_{12a}

stabilization of combinational hardware circuits with feedback loops as given in Theorem 1.1.

As a result, we conclude that 'causality of a program' is not a binary issue. Instead, there are different degrees of causality that can be implemented by a compiler that successively increases the complexity of the compilation but also the set of programs that can be compiled. Hence, causality is not only a property of a program alone, but also depends on the used code generation, since causality is a syntactic property rather than a semantic property. As a result, different code generators may differ in causality analysis although they produce logically equivalent code.

The proposals in Boussinot's work and in this paper therefore naturally impose the question whether there is a *semantic definition of causality*. Indeed, this semantic definition can be obtained by a maximal causality analysis, as we showed in [26]: by adding all prime implicants, we can show that the obtained equation system E_{max} for a program P can be used for this definition. In particular, the semantic definition of causality means that P is constructive iff E_{max} is constructive. This also yields the most powerful code generator in terms of the programs that it can handle in causality analysis. However, this requires to compute all prime implicants of the related Boolean functions, and for this reason, it is much more complex than the proposals given by Boussinot.

Finally, one may argue whether programs with a stronger notion of causality are reasonable at all. In [3] (page 36), it is argued that program P_{10} has a unique behavior so that reasonable code can, in principle, be generated for it. However, it is further argued that these programs are not good in the sense that information flows backwards in terms of program lines (not macro steps). However, we feel that it is not quite clear what 'backward information flow' means, since program P_{12} is rejected with the original definition of causality, while program P_{12a} in Figure 9 is accepted. To analyze program P_{12a}, one has to inspect the 'then' and 'else' branches in the same way as this has to be done for program P_{12}. Therefore, also in P_{12a} there is a backward flow of information, but nevertheless, P_{12a} is accepted, while P_{12} is rejected. If backward information flow should be forbidden at all, then we have to use functions f_0 and g_0 for the definitions in Sections 5.2 and 5.3, which is however not done in current compilers.

To sum up, we conclude that stronger notions of causality are reasonable, as long as they retain the relationship between stabilization of circuits, existence of dynamic schedules and constructive programs.

References

[1] Benveniste, A. and G. Berry, *The synchronous approach to reactive real-time systems*, Proceedings of the IEEE **79** (1991), pp. 1270–1282.

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

[3] Berry, G., *The constructive semantics of pure Esterel*, http://www-sop.inria.fr/esterel.org (1999).

[4] Berry, G., *The Esterel v5_91 language primer* (2000).

[5] Boussinot, F., *SugarCubes implementation of causality*, Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis Cedex (France) (1998).

[6] Bryant, R., D. Beatty and C.-J. Seger, *Formal hardware verification by symbolic ternary trajectory evaluation*, in: *Design Automation Conference (DAC)* (1991), pp. 397–402.

[7] Brzozowski, J. and C.-J. Seger, *Advances in asynchronous circuit theory part i*, Bulletin of the European association of Theoretical Computer Science (1990).

[8] Brzozowski, J. and C.-J. Seger, *Advances in asynchronous circuit theory part II*, Bulletin of the European Association of Theoretical Computer Science (1991).

[9] Brzozowski, J. and C.-J. Seger, "Asynchronous Circuits," Springer, 1995.

[10] Edwards, S., *Making cyclic circuits acyclic*, in: *Design Automation Conference (DAC)* (2003), pp. 159–162.

[11] Eichelberger, E., *Hazard detection in combinational and sequential switching circuits*, IBM Journal of Research and Development **9** (1965), pp. 90–99.

[12] Halbwachs, N., "Synchronous programming of reactive systems," Kluwer, 1993.

[13] Howard, W., "To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism," Academic, New York, 1980 pp. 479–490.

[14] Huffman, D., *Combinational circuits with feedback*, in: A. Mukhopadhyay, editor, *Recent Developments in Switching Theory* (1971), pp. 27–55.

[15] Kautz, W., *The necessity of closed circuit loops in minimal combinational circuits*, IEEE Transactions on Computers **C-19** (1970), pp. 162–166.

[16] Lassez, J.-L., V. Nguyen and E. Sonenberg, *Fixed point theorems and semantics. A folk tale.*, Information Processing Letters **14** (1982), pp. 112–116.

[17] Le Guernic, P., T. Gauthier, M. Le Borgne and C. Le Maire, *Programming real-time applications with SIGNAL*, IEEE **79** (1991), pp. 1321–1336.

[18] Malik, S., *Analysis of cyclic combinational circuits*, in: *Conference on Computer Aided Design (ICCAD)* (1993), pp. 618–625.

[19] Malik, S., *Analysis of cycle combinational circuits*, IEEE Transactions on Computer Aided Design **13** (1994), pp. 950–956.

[20] Mendler, M., *Timing analysis of combinational circuits in intuitionistic propositional logic*, Formal Methods in System Design **17** (2000), pp. 5–37.

[21] Mendler, M. and M. Fairtlough, *Ternary simulation: A refinement of binary functions or an abstraction of real-time behaviour*, in: *Workshop on Designing Correct Circuits (DCC)*, Electronic Workshops in Computing (1996).

[22] Rivest, R., *The necessity of feedback in minimal monotone combinational circuits*, IEEE Transactions on Computers **C-26** (1977), pp. 606–607.

[23] Schneider, K., *Embedding imperative synchronous languages in interactive theorem provers*, in: *Conference on Application of Concurrency to System Design (ACSD)* (2001), pp. 143–156.

[24] Schneider, K., *Proving the equivalence of microstep and macrostep semantics*, in: V. Carreño, C. Muñoz and S. Tahar, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, LNCS **2410** (2002), pp. 314–331.

[25] Schneider, K., J. Brandt and T. Schuele, *Causality analysis of synchronous programs with delayed actions*, in: *Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES)* (2004), pp. 179–189.

[26] Schneider, K., J. Brandt, T. Schuele and T. Tuerk, *Maximal causality analysis*, in: *Conference on Application of Concurrency to System Design (ACSD)* (2005).

[27] Shiple, T., "Formal Analysis of Synchronous Circuits," Ph.D. thesis, University of California at Berkeley (1996).

[28] Shiple, T., G. Berry and H. Touati, *Constructive analysis of cyclic circuits*, in: *European Design and Test Conference (EDTC)* (1996).

[29] Shiple, T., V. Singhal, R. Brayton and A. Sangiovanni-Vincentelli, *Analysis of combinational cycles in sequential circuits*, in: *Symposium on Circuits and Systems (ISCAS)*, 1996, pp. 592–595.

[30] Tarski, A., *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math **5** (1955), pp. 285–309.

[31] Yoeli, M. and S. Rinon, *Application of ternary algebra to the study of static hazards*, Journal of the ACM **11** (1964), pp. 84–97.