Maximal Causality Analysis

Klaus Schneider, Jens Brandt, Tobias Schuele, and Thomas Tuerk
Reactive Systems Group
Department of Computer Science
University of Kaiserslautern
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, which may lead to so-called causality cycles between actions and their trigger conditions. Algorithms to analyze the consistency of such cycles usually extend data types by an additional value to explicitly indicate unknown values. In particular, Boolean functions are thereby extended to ternary functions. However, a Boolean function usually has several ternary extensions, and the result of the causality analysis depends on the chosen ternary extension. In this paper, we show that there always is a maximal ternary extension that allows one to solve as many causality problems as possible. Moreover, we elaborate the relationship to hazard elimination in hardware circuits, and finally show how the maximal ternary extension of a Boolean function can be efficiently computed by means of binary decision diagrams.

1. Introduction

Synchronous languages [1, 2, 17, 23, 23] follow the *paradigm of perfect synchrony:* reactions (outputs) of the system respond immediately to actions (inputs) of the environment. This idealized programming model simplifies the semantics of the languages and leads to a simpler execution time analysis. Synchronous hardware circuits, as well as Mealy and Moore machines also follow the paradigm of perfect synchrony. However, since synchronous languages additionally allow programs to read their own outputs, mutual dependencies between actions and their trigger conditions may lead to so-called causality cycles. Such cycles may lead to inconsistencies so that no code can be generated. In many cases, however, causality cycles can be resolved and deterministic code can be generated. To this end, compilers have to analyze the causality of cyclic dependencies.

Causality cycles in hardware circuits (also called combinational cycles or feedback loops) have already been studied in the early seventies [19, 20, 32]. Kautz [20] and Rivest [32] proved that circuits with combinational cycles can be smaller than the smallest cycle-free implementations. For this reason, the introduction of combinational cycles has been recently proposed as a new strategy for logic minimization [30, 31]. Furthermore, causality cycles (called 'false paths' in this setting) occur in high-level synthesis of circuits by sharing common subexpressions [41], and are a major concern in the compilation of synchronous languages [2, 3, 5, 17, 34–36, 38]. As an example, consider the following equation (taken from [26]) where a system is to be implemented with large subsystems f and g:

$$y = \textbf{if } c \textbf{ then } f(g(x)) \textbf{ else } g(f(x)) \textbf{ end}$$

The cyclic equation system shown in Figure 1 requires only one instance of f and one instance of g (besides three multiplexors). In contrast, any acyclic version requires further instances of f and g.

$$\begin{cases} y &= \textbf{if } c \textbf{ then } y_f \textbf{ else } y_g \textbf{ end} \\ y_f &= f(x_f) \\ y_g &= g(x_g) \\ x_f &= \textbf{if } c \textbf{ then } y_g \textbf{ else } x \textbf{ end} \\ x_g &= \textbf{if } c \textbf{ then } x \textbf{ else } y_f \textbf{ end} \end{cases}$$

Figure 1. A Cyclic Equation System

In the above example, x and y may have arbitrary data types. In the following, however, we restrict our considerations to Boolean data types in order to circumvent additional problems concerning the termination of the causality analysis. Hence, given Boolean-valued input and output variables $\vec{x}$ and $\vec{y}$, the problem is to check whether a cyclic equation system $\vec{y} = \vec{f}(\vec{x}, \vec{y})$ has a unique solution for all inputs. There are a lot of equivalent views on (and thus applications of) causality analysis:

- Shiple [37] proved the equivalence to Brzozowski and Seger's timing analysis in the up-bounded inertial delay model [11]: Circuits derived from cyclic equation systems will stabilize for arbitrary gate delays iff the equation systems are causally correct. The algorithms used to analyze hardware circuits are based on a ternary interpretation of Boolean logic.
- Berry [3] pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic, since intuitionistic logic may be viewed in a ternary setting (1:provable, 0:disprovable, $\perp$: neither provable nor disprovable). Hence, he introduced the notion of constructive circuits [4, 27, 28, 38].
- By Berry's observation, the problem is equivalent to type-checking of certain functional programs due to the Curry-Howard isomorphism [18].
- Edwards reformulates the problem in that the existence of dynamic schedules must be guaranteed for the execution of mutually dependent actions [15].

Malik [26] was the first who presented algorithms for eliminating cycles in Boolean equation systems $\vec{y} = f(\vec{x}, \vec{y})$. He used a ternary extension of Boolean algebra as introduced by Yoeli and Rinon [43] and Eichelberger [16], and further refined by Brzozowski, Bryant, and Seger [6, 8–11] to analyze the propagation of signal values in these circuits. The computation of a solution $\vec{y}$ depending on the inputs $\vec{x}$ is then reduced to the computation of a fixpoint of the function $f_{\vec{x}}(\vec{y}) := \vec{f}(\vec{x}, \vec{y})$, where the inputs $\vec{x}$ are fixed. The existence of such fixpoints in the ternary domain is guaranteed by the Tarski-Knaster theorem [22, 42] (see also the next section), and the number of iterations is limited by the number of equations $|\vec{y}|$. Having computed the fixpoint $\vec{y} = f_{\vec{x}}(\vec{y})$ in a symbolical form, i.e., depending on the inputs $\vec{x}$, one has an equivalent acyclic equation system $\vec{y} = \vec{f'}(\vec{x})$. As this is done in a three-valued setting, it finally remains to check if all equations evaluate to Boolean values. It can be shown that computing the ternary fixpoint and checking whether it is a Boolean one is co-NP-complete [25, 37].

Although the acyclic version may require more operations than the original cyclic one [19, 20, 30–32], the elimination of cycles is a popular way for generating single-threaded sequential code from multi-threaded synchronous programs in that a causal order (i.e. a schedule [15]) to evaluate the right hand sides of the equation system is determined[1].

Malik's approach has been generalized by Shiple et al. to sequential circuits with cyclic output functions [37–39], and by Schneider et al. to arbitrary sequential circuits [34]. Besides a complete fixpoint computation, heuristics can be applied in a first instance to solve simple cases more efficiently [36]. Alternatives to the fixpoint computation were also considered: [12, 29] replaced the causality problem by the theoretically more difficult satisfiability problem and proposed new SAT solving techniques and temporal induction for its solution. However, causality analysis based on fixpoint computation is not only a heuristic for satisfiability checking, it moreover establishes a direct relationship between the causality of a program and its dynamic execution (stabilization of signals in circuits or existence of dynamic schedules in software). Hence, causality analysis may be viewed as a symbolic compile-time simulation of the program in order to guarantee its conflict-free execution.

All known procedures for causality analysis that are based on fixpoint computation require an extension of Boolean functions to a ternary domain. Although there are many ways how this can be done, none of the previous approaches considered the effect of different extensions. Instead, only basic Boolean operations like negation, conjunction, and disjunction are directly extended, and ternary extensions of other Boolean functions are obtained by composition of the ternary extensions of the basic functions. In [34], however, it has already been remarked that this is not optimal, and that different ternary extensions yield different results in causality analysis. This raises the question whether there is an optimal way to construct ternary extensions so that causality analysis can resolve as many causality cycles as possible.

In this paper, we answer this question to the positive: for every Boolean equation system, there is a (uniquely determined) maximal ternary extension that allows the transformation to an acyclic system, if this can be done by fixpoint computation at all. We show that this maximal ternary extension corresponds to the disjunction of all prime implicants, which gives a relationship to hazard elimination [16]. By this relationship, we derive a first algorithm for computing the maximal ternary extension, which, however, requires to compute all prime implicants of a Boolean function. We then present more efficient algorithms for computing the maximal ternary extension. In particular, we present an algorithm that can be easily implemented by means of binary ordered decision diagrams (BDDs).

The outline of the paper is as follows: in the next section, we formulate the problem of eliminating cycles in Boolean equation systems by reviewing Malik's procedure and its formal foundation based on Tarski's fixpoint theorem. In Section 3, we present the construction of the optimal ternary extension $\hat{f}$ and prove that it is the maximum of all ternary extensions of f. Moreover, we prove that an alternative construction of $\hat{f}$ is obtained by the disjunction of all prime implicants of f. After presenting our implementation and experimental results in Section 5, we conclude with a short summary.

[1] In the meantime, alternative compilation techniques [13, 14, 24] were proposed. However, these approaches still need causality analysis to guarantee the existence of a dynamic schedule for mutually dependent actions.

2. Elimination of Causality Cycles

In this section, we review the analysis of cyclic equation systems due to Malik [26], which is still the key to generalized algorithms for causality analysis [34, 37–39]. We describe Malik's method on a lattice theoretic background so that its correctness can be easily derived from the well-known fixpoint theorem of Tarski and Knaster [22, 42]. To this end, Boolean functions are extended to monotonic ternary functions, so that the existence of their fixpoints is guaranteed.

2.1. Formulation of the Problem

The problem is to decide for a given Boolean equation system of the following form whether it defines *unique outputs for all possible inputs*:

$$\begin{cases} y_1 = f_1(x_1, \ldots, x_m, y_1, \ldots, y_n) \\ \quad \vdots \\ y_n = f_n(x_1, \ldots, x_m, y_1, \ldots, y_n) \end{cases}$$

The arguments $x_1, \ldots, x_m$ are the inputs of the system, and $y_1, \ldots, y_n$ are the outputs. In the following, we make use of the shorthand vector notation $\vec{y} = \vec{f}(\vec{x}, \vec{y})$ for the above equation system. If we consider for every input $\vec{x}$ the function $\vec{f}_{\vec{x}}(\vec{y}) := \vec{f}(\vec{x}, \vec{y})$, it becomes obvious that the problem is to check whether this function has a unique fixpoint $\vec{y} = \vec{f}_{\vec{x}}(\vec{y})$. The Tarski-Knaster theorem as described in the next section presents an algorithm to decide this question. Moreover, it is possible to compute the fixpoint as a function $\vec{y} = \vec{f}'(\vec{x})$ depending only on the inputs $\vec{x}$ in order to generate an equivalent acyclic Boolean equation system (see e.g. [34]).

2.2. Fixpoint Theory

Fixpoint theory is well understood in theoretical computer science. In particular, the Tarski-Knaster theorem [22, 42] is often used to compute fixpoints of monotonic functions. For example, this theorem is fundamental to nearly all verification algorithms [33]. To apply this theorem and its related fixpoint iteration, we have to consider (complete) lattices.

A partially ordered set $(\mathcal{D}, \sqsubseteq)$ is a *lattice*, if all two-element sets $\{x, y\} \subseteq \mathcal{D}$ have suprema $\mathsf{sup}(\{x, y\})$ and infima $\mathsf{inf}(\{x, y\})$ in $\mathcal{D}$. A function $f : \mathcal{D} \to \mathcal{D}$ is monotonic, if for all $x, y \in \mathcal{D}$ with $x \sqsubseteq y$, we have $f(x) \sqsubseteq f(y)$. For monotonic functions $f : \mathcal{D} \to \mathcal{D}$ over a finite lattice $(\mathcal{D}, \sqsubseteq)$, it follows that $f(\mathsf{sup}(M)) = \mathsf{sup}(f(M))$ and $f(\mathsf{inf}(M)) = \mathsf{inf}(f(M))$ holds (i.e., monotonic functions over finite lattices are continuous). Moreover, in a finite lattice, $\mathsf{sup}(M)$ and $\mathsf{inf}(M)$ exist in $\mathcal{D}$ for every set $M \subseteq \mathcal{D}$. In particular, we write $\bot := \mathsf{inf}(\mathcal{D})$ and $\top := \mathsf{sup}(\mathcal{D})$ for the minimal and maximal element of $\mathcal{D}$, respectively.

Theorem 1 (Tarski/Knaster Fixpoint Theorem [22, 42]) *Let $(\mathcal{D}, \sqsubseteq)$ be a finite lattice and $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function. Then, f has fixpoints and the set of fixpoints even has a minimum $\check{x}$ and a maximum $\hat{x}$. Moreover, the least fixpoint $\check{x}$ of f can be computed by the iteration $p_0 := \bot$, $p_{i+1} := f(p_i)$, and the greatest fixpoint $\hat{x}$ of f can be computed by the iteration $q_0 := \top$, $q_{i+1} := f(q_i)$.*

2.3. Embedding Booleans in a Lattice

In order to apply the above theorem to causality analysis, we have to embed the Boolean values $\mathsf{0}$ and $\mathsf{1}$ in a lattice. In addition, we have to extend the considered Boolean functions $f_i(\vec{x}, \vec{y})$ to monotonic functions over this lattice. This can be achieved by extending $\mathbb{B} = \{\mathsf{0}, \mathsf{1}\}$ with the new elements $\bot$ and $\top$ to the set $\mathbb{F} := \{\bot, \mathsf{0}, \mathsf{1}, \top\}$. The partial order $\sqsubseteq$ on $\mathbb{F}$ is the reflexive-transitive closure of the relation where $\bot \sqsubseteq \mathsf{0}$, $\bot \sqsubseteq \mathsf{1}$, $\mathsf{0} \sqsubseteq \top$, and $\mathsf{1} \sqsubseteq \top$ holds. The partial order $\sqsubseteq$ naturally extends to vectors over $\mathbb{F}$, i.e., $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ holds for all components i.

In [34] it was shown that it suffices for causality analysis to consider least fixpoints. As a consequence, $\top$ is never needed for causality analysis, so that we can restrict our considerations to the semi-lattice $\mathbb{T} := \{\bot, \mathsf{0}, \mathsf{1}\}$, which is also a ternary algebra [11, 34]. This semi-lattice has been considered in many previous works including [16, 21, 26–28, 34, 37–39, 43] with different explanations of the third value. Our intuition is that the partial order is a measure for the information content, i.e., $x = \bot$ means that x could be either $\mathsf{0}$ or $\mathsf{1}$, but we do not yet know the Boolean value and therefore assign the value $\bot$. Causality analysis or ternary simulation proceeds in exactly this way and thereby computes a fixpoint according to the Tarski-Knaster theorem.

To this end, we have to extend a given Boolean function $f : \mathbb{B}^n \to \mathbb{B}^n$ to a monotonic function $g_f : \mathbb{T}^n \to \mathbb{T}^n$. In the following, we use the notion of a ternary extension:

Definition 1 (Ternary Extension) *A ternary function $g : \mathbb{T}^n \to \mathbb{T}^m$ is called a ternary extension of a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ if the following holds:*

- *$g(\vec{x}) = f(\vec{x})$ for all $\vec{x} \in \mathbb{B}^n$, and*
- *$g(\vec{x}_1) \sqsubseteq g(\vec{x}_2)$ for all $\vec{x}_1, \vec{x}_2 \in \mathbb{T}^n$ with $\vec{x}_1 \sqsubseteq \vec{x}_2$*

By our definition, every ternary extension $g : \mathbb{T}^n \to \mathbb{T}^n$ is a monotonic function and therefore, every ternary extension of this form has fixpoints that can be computed via the Tarski-Knaster iteration. However, a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^n$ may have many ternary extensions that can, in principle, all be used for the fixpoint computation. However, it is essential for causality analysis that the computed fixpoint is a Boolean one, and this may not hold for all ternary extensions of a Boolean function.

2.4. Causality Analysis by Fixpoint Iteration

As ternary extensions $f : \mathbb{T}^n \to \mathbb{T}^n$ are monotonic (by definition), it follows from the Tarski-Knaster theorem, that these functions have a least fixpoint in $\mathbb{T}^n$ that is computed by the simple iteration $\vec{y}_{i+1} := f(\vec{y}_i)$ starting with $\vec{y}_0 := (\bot, \ldots, \bot)$. If the least fixpoint $(\check{y}_1, \ldots, \check{y}_n)$ belongs to $\mathbb{B}^n$, it follows that the equation system has a Boolean solution. It is not difficult to see that this also implies the uniqueness of the solution [34].

Theorem 2 (Causality Analysis) *Given a Boolean vector function $f : \mathbb{B}^n \to \mathbb{B}^n$ and the least fixpoint $\check{\vec{y}}$ of an arbitrary ternary extension $g : \mathbb{T}^n \to \mathbb{T}^n$ of f, then the following propositions are equivalent:*

- $\check{\vec{y}} \in \mathbb{B}^n$
- $\vec{y} = f(\vec{y})$ *has only one solution in* $\mathbb{T}^n$

However, if $\check{\vec{y}} \notin \mathbb{B}^n$ holds, we know nothing about the existence of Boolean solutions of $\vec{y} = f(\vec{y})$.

In general, an equation system $\vec{y} = f(\vec{y})$ may have more than one solution in the three-valued domain. In particular, it may be the case that the least fixpoint $\check{\vec{y}}$ does not belong to $\mathbb{B}^n$, even if unique Boolean solutions $\vec{y}$ exist for all inputs. Moreover, it even depends on the used ternary extension if a unique Boolean solution is found. For example consider $f(x) := 0$ and two ternary extensions f_1 and f_2 of f:

x	$f(x)$	$f_1(x)$	$f_2(x)$
$\bot$	$-$	$\bot$	0
0	0	0	0
1	0	0	0

The least fixpoint of f_2 is 0 which is also the unique Boolean solution of $y = f(y)$. However, f_1 has two fixpoints and the fixpoint iteration yields its least fixpoint, namely $\bot$.

Thus, even if a unique Boolean solution exists, we may not be able to find this Boolean solution via fixpoint computation. While this is certainly a drawback of the procedure, the advantages are predominant: instead of checking (unique) satisfiability of the equation system, which is more difficult, we can compute the fixpoints with at most n iterations (this is the diameter of $\mathbb{T}^n$, i.e., the length of the largest chain in $\mathbb{T}^n$), and each iteration can be computed in linear time with respect to the size of the equation system. However, after this, we have to check whether the obtained fixpoint belongs to $\mathbb{B}^n$, which is a coNP-complete problem [25]. Checking whether an equation system $\vec{y} = \vec{f}(\vec{x}, \vec{y})$ has a unique solution $\vec{y}$ for all inputs $\vec{x}$, requires to prove the validity of the formula $\forall\vec{x}.\exists_1\vec{y}.\vec{y} = \vec{f}(\vec{x}, \vec{y})$, which is probably more difficult than coNP-complete (currently, we do not know a precise complexity class of the problem).

3. Maximal Ternary Extensions

3.1. Standard Ternary Extension

In the previous section, we showed that more than one ternary extension of a Boolean function may exist and that the success of causality analysis depends on the chosen extension. However, there is only one possible extension $\widetilde{\neg}$ for the negation $\neg$, and there are four extensions for conjunction and disjunction. Usually, the following ternary extensions $\widetilde{\neg}$, $\widetilde{\wedge}$ and $\widetilde{\vee}$ of $\neg$, $\wedge$ and $\vee$ are chosen:

$\widetilde{\wedge}$	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

$\widetilde{\vee}$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

x	$\widetilde{\neg}x$
$\bot$	$\bot$
0	1
1	0

The above three-valued operations have already been used by Kleene [21] and in many other publications [11, 16, 21, 26–28, 34, 37–39, 43] with different interpretations of the third value. In [34], it was argued that these ternary extensions are maximal, and therefore they are best suited for causality analysis (since their fixpoints are also maximal, and thus more probably Boolean).

As every Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ can be represented by a propositional formula with n variables with the operators $\neg$, $\wedge$, and $\vee$, it follows that we can represent every Boolean function as a composition of these basic Boolean functions. Moreover, as monotonic functions are closed under function composition, a ternary extension $g_f : \mathbb{T}^n \to \mathbb{T}$ of a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ is directly obtained by replacing $\neg$, $\wedge$, and $\vee$ with the above ternary extensions $\widetilde{\neg}$, $\widetilde{\wedge}$, and $\widetilde{\vee}$, respectively.

The ternary function $g_f : \mathbb{T}^n \to \mathbb{T}$ that is thereby obtained is called the *standard ternary extension* of f. Note, however, that g_f depends not only on the Boolean function, but also on its *representation by a particular propositional formula.* However, the representation of f by propositional formulas is not unique: It can be easily shown (see example in Section 3.5), that *different ternary extensions are obtained for different propositional representations of f*. Moreover, these different choices influence the success of causality analysis. Hence, this definition makes causality a syntactical property [34]: *There are logically equivalent Boolean equation systems where one is causally correct, but the other one is not*. In [35], we therefore showed how code generation can be improved so that more programs can be successfully handled in causality analysis. Having a closer look at the improvements that are discussed in [35] shows that they are all special cases of the maximal causality analysis presented in this paper. Their advantage is, however, that they can be implemented more efficiently.

3.2. Maximal Ternary Extension

In this section we will show that among the ternary extensions, there always is a best choice: the *maximal ternary extension*. We show that every Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ has a maximal ternary extension. Using this extension, we obtain the best results for causality analysis in that as many cyclic equation systems as possible can be solved. As the maximal ternary extension is moreover defined w.r.t. the semantics, *causality in this sense is no longer a syntactic issue*. This means that the causality does no longer depend on the syntax of a propositional formula that is used as the representation of a Boolean function f.

Definition 2 (Maximal Ternary Extension) *A ternary extension $g : \mathbb{T}^n \to \mathbb{T}^m$ of a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ is called the maximal ternary extension of f if $h(\vec{x}) \sqsubseteq g(\vec{x})$ holds for all ternary extensions $h : \mathbb{T}^n \to \mathbb{T}^m$ of f and for all $\vec{x} \in \mathbb{T}^n$.*

Obviously, there is at most one maximal ternary extension of a Boolean function. Since the partial order $\sqsubseteq$ on $\mathbb{T}^n$ is defined component-wise, the function $g : \mathbb{T}^n \to \mathbb{T}^m$ with $g = (g_1, \ldots, g_m)$ is a maximal ternary extension of the function $f : \mathbb{B}^n \to \mathbb{B}^m$ with $f = (f_1, \ldots, f_m)$, iff $g_i : \mathbb{T}^n \to \mathbb{T}$ is the maximal ternary extension of $f_i : \mathbb{B}^n \to \mathbb{B}$ (for all i). Therefore, it is sufficient to consider only maximal ternary extensions of Boolean functions $f : \mathbb{B}^n \to \mathbb{B}$.

Theorem 3 (Maximal Ternary Extension) *Let $g : \mathbb{T}^n \to \mathbb{T}^n$ be the maximal ternary extension of a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^n$ and let $h : \mathbb{T}^n \to \mathbb{T}^n$ be an arbitrary ternary extension of f. If causality analysis by fixpoint iteration using h succeeds, causality analysis using g also succeeds.*

Proof. Let $\vec{y_h}$ be the least fixpoint of h. Then, there exists an $i \in \mathbb{N}$ with $h^i(\bot, \ldots, \bot) = \vec{y_h}$. As g is the maximal ternary extension, $\vec{y_h} = h^i(\bot, \ldots, \bot) \sqsubseteq g^i(\bot, \ldots, \bot) =: \vec{y_g}$ holds. We know that $\vec{y_h}$ is Boolean, since causality analysis using h succeeds. Hence, $\vec{y_h} \sqsubseteq \vec{y_g}$ implies $\vec{y_h} = \vec{y_g}$. Thus, $\vec{y_g}$ is Boolean and causality analysis using g succeeds. □

3.3. First Algorithm to Compute the Maximal Ternary Extension

In this section, we present a first algorithm for the construction of a ternary extension $\hat{f}$ for a Boolean function f. After that, we will prove that $\hat{f}$ is the maximal ternary extension of f. To define $\hat{f}$, the following definition is useful:

Definition 3 (Degree of a Ternary Vector) *The degree $\mathsf{degree}(\vec{x})$ of a ternary vector $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{T}^n$ is the number of non-Boolean components x_i, that is the number of x_i's which are $\bot$.*

Note that $\vec{x} \sqsubseteq \vec{y}$ holds iff for all $i \in \{1, \ldots, n\}$, we have $x_i = y_i$ or $x_i = \bot$. Hence, it follows that $\vec{x} \sqsubseteq \vec{y}$ implies $\mathsf{degree}(\vec{x}) \geq \mathsf{degree}(\vec{y})$ (the reverse is not the case). Note that the ternary vectors of degree 0 are the Boolean vectors. This property is used in the following definition of the maximal ternary extension $\hat{f}$ of a Boolean function f:

Definition 4 (Maximal Ternary Extension) *For every Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, we define a ternary function $\hat{f}$ recursively as follows:*

- *for $\vec{x} \in \mathbb{T}^n$ with $\mathsf{degree}(\vec{x}) = 0$, define $\hat{f}(\vec{x}) = f(\vec{x})$*
- *for $\vec{x} \in \mathbb{T}^n$ with $\mathsf{degree}(\vec{x}) = m + 1$, define*

$$\hat{f}(\vec{x}) = \mathsf{inf}\left(\{\hat{f}(\vec{y}) \mid \vec{x} \sqsubseteq \vec{y} \wedge \mathsf{degree}(\vec{y}) = m\}\right)$$

Note that $\vec{x} \sqsubseteq \vec{y}$ and $\mathsf{degree}(\vec{x}) = \mathsf{degree}(\vec{y}) + 1$ imply that the vectors $\vec{x}$ and $\vec{y}$ differ in exactly one component x_j, i.e., we have $x_i = y_i$ for all $i \in \{1, \ldots, n\} \setminus \{j\}$ and $y_j = \bot$ and $x_j \in \mathbb{B}$. Hence, to determine $\hat{f}(\vec{x})$, we have to inspect all previously determined function values $\hat{f}(\vec{y})$ where $\vec{x}$ and $\vec{y}$ are as described above.

Because of the construction of $\hat{f}$, it is obvious that $\hat{f}(\vec{x}) = f(\vec{x})$ holds for all Boolean valued $\vec{x}$. We can also prove the monotonicity: consider $\vec{x}, \vec{y} \in \mathbb{T}^n$ with $\vec{x} \sqsubseteq \vec{y}$. It follows that $\mathsf{degree}(\vec{x}) \geq \mathsf{degree}(\vec{y})$ holds, which implies by definition of $\hat{f}$ that $\hat{f}(\vec{x}) \sqsubseteq \hat{f}(\vec{y})$ holds. Using this, it is easily seen that we could alternatively define for $\vec{x} \notin \mathbb{B}^n$:

$$\hat{f}(\vec{x}) = \mathsf{inf}\left(\{\hat{f}(\vec{y}) \mid \vec{x} \sqsubseteq \vec{y}\}\right)$$

However, this does not directly yield an algorithm for the construction of $\hat{f}$. For this reason, we decided to use the refined version above (which only has to inspect already available function values). By the above results, we already conclude that $\hat{f}$ is a ternary extension of f. It can moreover be seen that it is the maximal one:

Theorem 4 *The ternary function $\hat{f}$ is the maximal ternary extension of f.*

Proof. Let g be an arbitrary ternary extension of f. We must show that $g(\vec{x}) \sqsubseteq \hat{f}(\vec{x})$ holds for all inputs $\vec{x} \in \mathbb{T}^n$. We prove this by induction on the degree of $\vec{x}$. In case $\mathsf{degree}(\vec{x}) = 0$ the input $\vec{x}$ is Boolean and therefore $g(\vec{x}) = f(\vec{x}) = \hat{f}(\vec{x})$ holds. So let $\mathsf{degree}(\vec{x}) = n+1$ hold. Since g is a ternary extension, $g(\vec{x}) \sqsubseteq g(\vec{y})$ holds for all $\vec{y}$ with $\vec{x} \sqsubseteq \vec{y}$. We already know $g(\vec{y}) \sqsubseteq \hat{f}(\vec{y})$ for all $\vec{y}$ with $\mathsf{degree}(\vec{y}) = n$ by induction hypothesis. Thus, $g(\vec{x}) \sqsubseteq g(\vec{y}) \sqsubseteq \hat{f}(\vec{y})$ for all $\vec{y}$ with $\vec{x} \sqsubseteq \vec{y}$ and $\mathsf{degree}(\vec{y}) = n$ follows. Therefore $g(\vec{x}) \sqsubseteq \mathsf{inf}\left(\{\hat{f}(\vec{y}) \mid \vec{x} \sqsubseteq \vec{y} \wedge \mathsf{degree}(\vec{y}) = n\}\right) = \hat{f}(\vec{x})$ holds. □

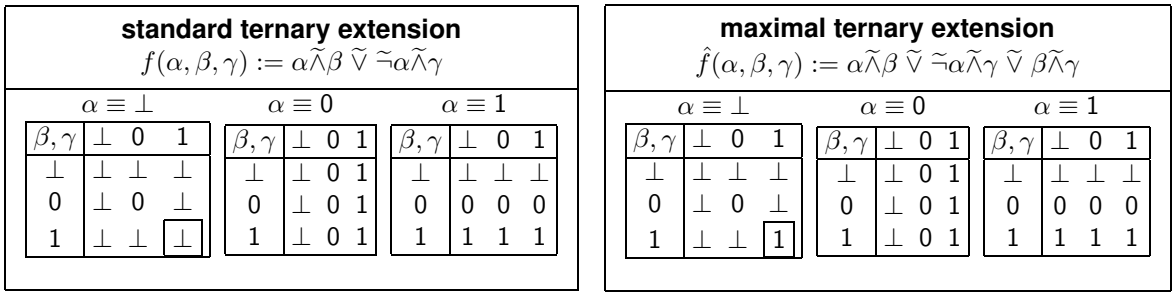

Figure 2. Standard and Maximal Ternary Extensions of the 'if-then-else' Function.

3.4. Second Algorithm to Compute the Maximal Ternary Extension

Eichelberger considered ternary extensions in [16] in order to eliminate hazards in hardware circuits. To define ternary extensions, he distinguished between basic gates and combined circuits. His construction for basic gates is different to the one given above, but it yields the same ternary extension, which can be seen by the following lemma (which is Eichelberger's definition):

Lemma 1 (Eichelberger's Ternary Extension) *Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ and its maximal ternary extension $\hat{f} : \mathbb{T}^n \to \mathbb{T}$. Then, abbreviate for $\vec{x} \in \mathbb{T}^n$:*

$$\mathsf{Cube}(\vec{x}) := \{\vec{y} \in \mathbb{B}^n \mid \vec{x} \sqsubseteq \vec{y}\}$$

Using this abbreviation, the following holds for all $\vec{x} \in \mathbb{T}^n$:

- *$\hat{f}(\vec{x}) = 0$ iff we have $f(\vec{y}) = 0$ for all $\vec{y} \in \mathsf{Cube}(\vec{x})$.*
- *$\hat{f}(\vec{x}) = 1$ iff we have $f(\vec{y}) = 1$ for all $\vec{y} \in \mathsf{Cube}(\vec{x})$.*
- *$\hat{f}(\vec{x}) = \bot$ iff there are $\vec{y}_1, \vec{y}_2 \in \mathsf{Cube}(\vec{x})$ with $f(\vec{y}_1) = 0$ and $f(\vec{y}_2) = 1$.*

Hence, $\hat{f}(\vec{x}) = \mathsf{inf}\,(\{f(\vec{y}) \mid \vec{y} \in \mathsf{Cube}(\vec{x})\})$.

Eichelberger used the above facts to define $\hat{f}$ without having the wish to construct a maximal ternary extension in our sense. Instead, a vector $\vec{x} \in \mathbb{T}^n$ is identified with $\mathsf{Cube}(\vec{x})$. Thus, Eichelberger used $\bot$ as a place-holder for the Boolean values, while we use it as a measure of information content. Of course, the above lemma tells us that both intuitions yield the same ternary function. Therefore, we can list another characterization of $\hat{f}$:

Theorem 5 (Prime Implicant Theorem) *The maximal ternary extension $\hat{f} : \mathbb{T}^n \to \mathbb{T}$ of a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ is obtained by replacing $\neg$, $\wedge$, and $\vee$ by $\widetilde{\neg}$, $\widetilde{\wedge}$, and $\widetilde{\vee}$ in the disjunction of all prime implicants of f.*

Proof. Prime implicants can be represented by minimal vectors $\vec{x} \in \mathbb{T}^n$ so that $\mathsf{Cube}(\vec{x})$ contains the assignments that satisfy the prime implicant. In this role $x_i = \bot$ means that the value of x_i does not care. Once this is seen, the above theorem follows almost from the previous lemma: In particular, if a prime implicant is missing, we can find $\vec{y}_1, \vec{y}_2 \in \mathsf{Cube}(\vec{x})$ with $f(\vec{y}_1) = 0$ and $f(\vec{y}_2) = 1$, and thus would have $\hat{f}(\vec{x}) = \bot$. □

The above result gives us a second algorithm to compute the maximal ternary extension $\hat{f}$ of a given Boolean function f.

3.5. Example

As we have seen, the choice of a ternary extension for a given Boolean function matters for causality analysis, and the maximal ternary extension is the best choice to solve as many problems as possible. In this section, we will illustrate this by an example that often occurs in the compilation of synchronous programs. To this end, consider the 'if-then-else' operator as a simple Boolean function $f(\alpha, \beta, \gamma)$ with three arguments. It can be represented with the following propositional formula:

$$f(\alpha, \beta, \gamma) :\equiv \alpha \wedge \beta \vee \neg\alpha \wedge \gamma.$$

Figure 2 shows the standard and maximal ternary extensions of this Boolean function. According to the results of the previous section, the maximal ternary extension is the standard ternary extension of the disjunction of all prime implicants of f, i.e., the standard extension of $\alpha \wedge \beta \vee \neg\alpha \wedge \gamma \vee \beta \wedge \gamma$. As can be seen, the two possible extensions of f differ in $(\alpha, \beta, \gamma) = (\bot, 1, 1)$. This allows $\hat{f}$ to select β if $\beta = \gamma$ holds, regardless of the value of α. As a consequence, the causality analysis given in Figure 3 depends on the ternary extension (note that $\bot\widetilde{\wedge}x\widetilde{\vee}\bot \equiv \bot$ and $\bot\widetilde{\wedge}x\widetilde{\vee}x \equiv x$ holds).

standard extension	maximal extension
$\begin{cases} y = y \wedge x \vee \neg y \wedge z \\ z = x \end{cases}$	$\begin{cases} y = y \wedge x \vee \neg y \wedge z \vee x \wedge z \\ z = x \end{cases}$

	0	1	2
y	$\bot$	$\bot$	$\bot$
z	$\bot$	x	x

	0	1	2	3
y	$\bot$	$\bot$	x	x
z	$\bot$	x	x	x

Figure 3. Causality Analysis Depends on Ternary Extension.

4. Implementation by Dual-Rail Encoding

In this section, we show how causality analysis can be implemented symbolically, e.g. by means of BDDs. The method used here has been proposed by Bryant for ternary simulation of MOS transistor circuits [6]. For reasons of efficiency, we directly encode the fact that input and state variables are always Boolean values. This fact can be encoded easily, which saves a lot of propositional variables and makes our analysis more efficient.

Similar to Bryant's work, we start with a dual-rail encoding where *sets over the lattice $\mathbb{F}$ are encoded by a pair of propositional formulas*. This representation is based on the following encoding of $\mathbb{F}$ by $\mathbb{B}^2$:

Definition 5 (Dual-Rail Encoding) *For the encoding of the values of $\mathbb{F}$ by values of $\mathbb{B}^2$, we use the following bijective mapping $\epsilon : \mathbb{B}^2 \to \mathbb{F}$:*

$x \in \mathbb{B}^2$	$\epsilon(x) \in \mathbb{F}$
$(0,0)$	$\bot$
$(0,1)$	0
$(1,0)$	1
$(1,1)$	$\top$

Using dual-rail encoding, we can encode the values of $\mathbb{F}$ and all desired operations on $\mathbb{F}$ by corresponding Boolean values and operations, respectively. In particular, we can represent a ternary function $f : \mathbb{T}^n \to \mathbb{T}$ by two Boolean functions $(g(\vec{u},\vec{v}), h(\vec{u},\vec{v}))$ if the following equation holds:

$$\epsilon(g(\vec{u},\vec{v}), h(\vec{u},\vec{v})) := f(\epsilon(\vec{u},\vec{v}))$$

For example, the maximal ternary extensions $\widetilde{\neg}$, $\widetilde{\wedge}$ and $\widetilde{\vee}$ of $\neg$, $\wedge$ and $\vee$ discussed in Section 3 are encoded by $\neg_2$, $\wedge_2$, and $\vee_2$, that are defined as follows:

- $\neg_2(x_1, x_2) := (x_2, x_1)$
- $(x_1, x_2) \wedge_2 (y_1, y_2) := (x_1 \wedge y_1, x_2 \vee y_2)$
- $(x_1, x_2) \vee_2 (y_1, y_2) := (x_1 \vee y_1, x_2 \wedge y_2)$

Thus, canonical normal forms for propositional logic like binary decision diagrams (BDDs) [7, 40] lead to canonical normal forms for functions on $\mathbb{T}$. To obtain an efficient implementation, we can therefore easily use BDDs for representing and manipulating these formulas.

4.1. Standard Ternary Extension

In the following, we are especially interested in representations of ternary extensions. Recall that the Boolean functions we are interested in are given by an equation system of the following form

$$\begin{cases} y_1 &= f_1(x_1, \ldots, x_m, y_1, \ldots, y_n) \\ &\vdots \\ y_n &= f_n(x_1, \ldots, x_m, y_1, \ldots, y_n) \end{cases}$$

As motivated in Section 3.2, it is sufficient to consider all f_i individually in order to build a ternary extension of $\vec{f}$. Furthermore, it is advantageous to exploit the fact that input variables x_i are always Booleans. Figure 4 shows the function $\mathsf{Rails}_{\mathsf{std}}$ that computes the standard ternary extension of a propositional formula $f : \mathbb{B}^{m+n} \to \mathbb{B}$ and a set of output variables $\mathcal{Y}$ (containing $y_1, \ldots, y_n$): $\mathsf{Rails}_{\mathsf{std}}(\mathcal{Y}, f)$ yields two propositional formulas (g, h) that are the rails of f. For notational convenience, we encode a variable $y \in \mathcal{Y}$ by the two Boolean variables $(y.r1, y.r2)$ denoting the two rails of y. The correctness of $\mathsf{Rails}_{\mathsf{std}}$ is obvious, since the standard ternary extension is defined by replacing $\neg$, $\wedge$ and $\vee$ with $\widetilde{\neg}$, $\widetilde{\wedge}$ and $\widetilde{\vee}$, and we have already seen encodings of $\widetilde{\neg}$, $\widetilde{\wedge}$ and $\widetilde{\vee}$.

Note that for variables $x \notin \mathcal{Y}$, we return the pair $(x, \neg x)$ which means that we have got a Boolean value. This is how we encode the knowledge that inputs and state variables are Booleans. As a result, we only have to generate copies of the output variables, and therefore save a lot of variables to minimize the BDD sizes.

4.2. Maximal Ternary Extension

It is also possible to directly define the rails g, h of the maximal ternary extension $\hat{f}$ in terms of the Boolean function $f : \mathbb{B}^{m+n} \to \mathbb{B}$:

- $\mathsf{InCube}(\vec{y}, \vec{u}, \vec{v}) := \bigwedge_{i=1}^{n}(u_i \to y_i) \wedge \bigwedge_{i=1}^{n}(v_i \to \neg y_i)$
- $g(\vec{x}, \vec{u}, \vec{v}) := \forall \vec{y}.\ \mathsf{InCube}(\vec{y}, \vec{u}, \vec{v}) \to f(\vec{x}, \vec{y})$
- $h(\vec{x}, \vec{u}, \vec{v}) := \forall \vec{y}.\ \mathsf{InCube}(\vec{y}, \vec{u}, \vec{v}) \to \neg f(\vec{x}, \vec{y})$

The correctness of this encoding follows from Lemma 1. $\mathsf{InCube}(\vec{y}, \vec{u}, \vec{v})$ holds iff the Boolean vector $\vec{y}$ belongs to the set of ternary vectors that are greater than or equal to the ternary vector encoded by $(\vec{u}, \vec{v})$ (i.e., if $\epsilon(\vec{y}, \neg\vec{y}) \sqsubseteq \epsilon(\vec{u}, \vec{v})$ holds.)

```
function Rails_std(𝒴, f)
  case f of
    ¬f1    : (g, h) := Rails_std(𝒴, f1);
             return (h, g);
    f1 ∧ f2: (g1, h1) := Rails_std(𝒴, f1);
             (g2, h2) := Rails_std(𝒴, f2);
             return (g1 ∧ g2, h1 ∨ h2);
    f1 ∨ f2: (g1, h1) := Rails_std(𝒴, f1);
             (g2, h2) := Rails_std(𝒴, f2);
             return (g1 ∨ g2, h1 ∧ h2);
    0      : return (0, 1);
    1      : return (1, 0);
    x      : if x ∈ 𝒴 then
               return (x.r1, x.r2)
             else return (x, ¬x) end;
  end case
end function
```

Figure 4. Computing the Dual-Rails of the Standard Ternary Extension

```
function Rails_max(𝒴, f)
  C := ⋀_{i=1}^{n} (u_i → y_i) ∧ ⋀_{i=1}^{n} (v_i → ¬y_i);
  g := ∀y⃗. C → f;
  h := ∀y⃗. C → ¬f;
  return (g, h)
end function
```

Figure 5. Computing the Dual-Rails of the Maximal Ternary Extension with BDDs

According to the dual-rail encoding, it follows that $g(\vec{x},\vec{u},\vec{v}) = 1$ holds iff $\hat{f}(\vec{x},\epsilon(\vec{u},\vec{v})) = 1$ holds, which is according to Lemma 1 equivalent to $f(\vec{x},\vec{y}) = 1$ for all Boolean vectors $\vec{y}$ with $\mathsf{InCube}(\vec{y},\vec{u},\vec{v})$. Analogously, $h(\vec{x},\vec{u},\vec{v}) = 1$ holds iff $\hat{f}(\vec{x},\epsilon(\vec{u},\vec{v})) = 0$ holds, which is according to Lemma 1 equivalent to $f(\vec{x},\vec{y}) = 0$ for all Boolean vectors $\vec{y}$ with $\mathsf{InCube}(\vec{y},\vec{u},\vec{v})$.

Hence, it is possible to directly implement the maximal ternary extension $\hat{f}$ of a given Boolean function f in terms of two Boolean functions g, h via dual-rail encoding and Boolean quantification. In practice, a slightly improved method is used to get smaller BDDs: Instead of considering all output variables $\vec{y}$ of f, only output variables that are used in f are taken into account. Further optimizations result from the fact that the value $\top$ can never occur .

4.3. Fixpoint Iteration

Using dual-rail encoding, it is straightforward to implement the fixpoint computations as known from symbolic model checking (see [33]). This can be done for the computation of the acyclic equation system (cf. Figure 6) as well as for reachability analysis, which is a standard problem in model checking. For the latter, we must verify that for all paths starting in an initial state, the rails of each output variable are always complementary. In this case, the outputs are Booleans, and we can use their first rails for code generation.

```
function MakeAcyclic(y⃗ = f⃗(x⃗, y⃗))
  𝒴 := {y_1, ..., y_|y⃗|};
  (g⃗, h⃗) := Rails_max(𝒴, Φ⃗);
  g⃗_new := (0, ..., 0);
  h⃗_new := (0, ..., 0);
  do
    g⃗_old := g⃗_new;
    h⃗_old := h⃗_new;
    g⃗_new := [g⃗]^{g⃗_old, h⃗_old}_{u⃗,v⃗} ;
    h⃗_new := ⌈h⃗⌉^{g⃗_old, h⃗_old}_{u⃗,v⃗} ;
  while (g⃗_new ≠ g⃗_old) ∨ (h⃗_new ≠ h⃗_old)
  return (g⃗_new, h⃗_new)
end
```

Figure 6. Causality Analysis with BDDs

5. Experimental Results

In order to evaluate the impact of maximal ternary extensions, we randomly generated equation systems with n output and $2n$ input variables for particular numbers n. The right-hand sides of the equations were formulas built of $\neg$, $\wedge$, and $\vee$ with at most 20 symbols. Limiting the size of the formulas this way, and choosing twice as many input variables as output variables gave a reasonable setting for finding solvable acyclic equation systems. Then, we used the standard and maximal ternary extensions to check whether the equation systems have unique Boolean solutions. For this purpose, we used the implementations as described above. Values measured in each case were:

- the number of resolvable equation systems
- the number of iterations and the required runtime
- the average size of the BDDs encoding the equation systems.

var. n	tested examples	uniquely solvable	standard solved	standard iterations	standard time (ms)	standard size	maximal solved	maximal iterations	maximal time (ms)	maximal size
10	5000	604	539	23866	5081	109	567	23584	8393	122
15	5000	443	397	30500	27956	172	414	30204	39409	195
20	5000	281	256	36931	254631	237	266	36643	329478	270

Table 1. Experimental Results

Table 1 lists the results of the experiments for 5000 equations for $n \in \{10, 15, 20\}$. Additionally, we list in the third column how many examples had a unique Boolean solution.

As expected, the maximal ternary extension allows us to solve more examples than the standard extension. Furthermore, it needs less iterations, since it converges faster. However, causality analysis with the maximal ternary extension requires more computation time: Firstly, the computation of the maximal ternary extension requires additional time, and secondly, the BDDs encoding the maximal ternary extension are bigger than those encoding the standard ternary extension.

Figure 7 compares the efficiency of both extensions. In the diagram, each cross represents one example, where the coordinates are the runtimes for the causality analysis with the maximal and standard ternary extensions, respectively. As can be seen, the overhead due to the maximal ternary extension is acceptable.

The experiments show that the maximal ternary extension needs about 33.5% more time than causality analysis with the standard extension (see Figure 7). Thus, the benefits of the maximal ternary extension (in solving more examples) outweigh the worse execution time.

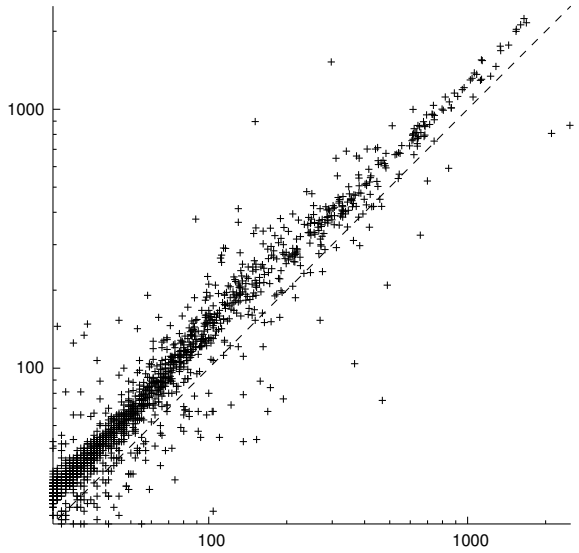

Figure 7. Time Using Maximal and Standard Ternary Extensions

6. Conclusions

In this paper, we considered a fundamental problem [3, 5, 36, 38] that appears in the compilation of perfectly synchronous systems [2, 17], namely the analysis and elimination of cyclic dependencies between actions and their trigger conditions. State of the art techniques solve this problem by computing the least fixpoint of the ternary extension of the Boolean equation system that is obtained by replacing $\neg$, $\wedge$, and $\vee$ by their ternary extensions $\widetilde{\neg}$, $\widetilde{\wedge}$, and $\widetilde{\vee}$. In this paper, we improved this causality analysis by showing that every Boolean function has a maximal ternary extension that should be chosen for the fixpoint iteration. Moreover, we revealed the relationship to hazard detection, and therefore were able to present several algorithms to compute the maximal ternary extensions. The additional effort to compute the maximal ternary extension is acceptable, but allows one to compile more programs.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

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

[4] G. Berry. The Esterel v5_91 language primer, June 2000.

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

[6] R. Bryant. A switch level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[7] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–

691, August 1986.

[8] R. Bryant, D. Beatty, and C.-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Design Automation Conference (DAC)*, pages 397–402, San Francisco, California, USA, 1991. ACM.

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

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

[11] J. Brzozowski and C.-J. Seger. *Asynchronous Circuits*. Springer, 1995.

[12] K. Claessen. Safety property verification of cyclic synchronous circuits. In *Synchronous Languages, Applications, and Programming (SLAP)*, volume 88 of *ENTCS*, Porto, Portugal, 2003. Elsevier.

[13] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. In *Synchronous Languages, Applications, and Programming (SLAP)*, volume 65 of *ENTCS*, Grenoble, France, 2002. Elsevier.

[14] S. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, 2003.

[15] S. Edwards. Making cyclic circuits acyclic. In *Design Automation Conference (DAC)*, pages 159–162, Anaheim, California, USA, 2003. ACM.

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

[17] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[18] W. Howard. *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, chapter The formulas-as-types notion of construction, pages 479–490. Academic, New York, 1980.

[19] D. Huffman. Combinational circuits with feedback. In A. Mukhopadhyay, editor, *Recent Developments in Switching Theory*, pages 27–55. Academic Press, 1971.

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

[21] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[22] J.-L. Lassez, V. Nguyen, and E. Sonenberg. Fixed point theorems and semantics. A folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[23] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *IEEE*, 79(9):1321–1336, 1991.

[24] J. Lukoschus and R. von Hanxleden. Removing cycles in Esterel programs. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[25] S. Malik. Analysis of cyclic combinational circuits. In *Conference on Computer Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, November 1993. IEEE Computer Society.

[26] S. Malik. Analysis of cycle combinational circuits. *IEEE Transactions on Computer Aided Design*, 13(7):950–956, July 1994.

[27] M. Mendler. Timing analysis of combinational circuits in intuitionistic propositional logic. *Formal Methods in System Design*, 17(1):5–37, 2000.

[28] M. Mendler 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, Bastad, Sweden, 1996. Springer.

[29] K. Namjoshi and R. Kurshan. Efficient analysis of cyclic definitions. In N. Halbwachs and D. Peled, editors, *Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 394–405, Trento, Italy, 1999. Springer.

[30] M. Riedel and J. Bruck. Cyclic combinational circuits: Analysis for synthesis. In *International Workshop on Logic and Synthesis (IWLS)*, Laguna Beach, California, 2003.

[31] M. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *Design Automation Conference (DAC)*, pages 163–168, Anaheim, California, USA, 2003. ACM.

[32] R. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers*, C-26(6):606–607, 1977.

[33] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[34] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES)*, pages 179–189, Washington D.C., USA, September 2004. ACM.

[35] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[36] E. Sentovich. Quick conservative causality analysis. In *International Symposium on System Synthesis (ISSS)*, pages 2–8, Antwerp, Belgium, 1997. IEEE Computer Society.

[37] T. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, 1996.

[38] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC)*, Paris, France, 1996. IEEE Computer Society.

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

[40] F. Somenzi. Binary decision diagrams, 1999.

[41] L. Stok. False loops through resource sharing. In *Conference on Computer Aided Design (ICCAD)*, pages 345–348. IEEE Computer Society, 1992.

[42] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

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