Causality Analysis of Synchronous Programs with Delayed Actions

K. Schneider, J. Brandt, and T. Schuele

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

ABSTRACT

Synchronous programs are well-suited for the implementation of real-time embedded systems. However, their compilation is difficult due to the paradigm that microsteps are executed in zero time. This can yield cyclic dependencies that must be resolved to generate single-threaded code. State of the art techniques are based on a fixpoint computation at compile time that ‘simulates’ the microstep execution. However, existing procedures do not consider delayed actions that have been recently introduced in synchronous languages. In this paper, we show that the analysis of programs with delayed actions can be performed by two fixpoint computations, one for the initialization and one for the transitions of the system. Moreover, we discuss an implementation using BDDs that is based on dual rail encoding.

Categories and Subject Descriptors

B.6 [Hardware]: Logic Design; D.3 [Software]: Programming Languages

General Terms

Algorithms, Design, Languages, Theory, Verification.

Keywords

Synchronous Languages, Causality, Fixpoints, Ternary Logic

1. INTRODUCTION

With the advent of synchronous languages [1, 15], a new paradigm for programming reactive real-time systems has been established. In this paradigm, the execution of a program is divided into macrosteps [16] that consist of finitely many microsteps. From a programmer’s view, the execution of a microstep requires no time, whereas the macrosteps all require the same logical amount of time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES’04, September 22–25, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-890-3/04/0009 ...$5.00.

The introduction of micro- and macrosteps is not only a convenient programming model, it is also the key to generate deterministic single-threaded code from multi-threaded synchronous programs. As a result, synchronous programs can be executed on ordinary microcontrollers without complex operating systems and therefore allow precise predictions of worst-case execution times [20, 21, 35, 36]. As another advantage of the synchronous programming paradigm, the translation to synchronous circuits is straightforward [2, 3, 28, 29, 33]. For these reasons, synchronous languages lend themselves naturally for hardware-software codesign. Moreover, their formal semantics is attractive for reasoning about program semantics and program correctness [3, 30, 31, 33].

While convenient for the programmer, the synchronous programming paradigm challenges the compilers [12]. Before code generation can be performed, the compilers have to solve schizophrenia problems [3, 33, 34, 43] and causality problems [3, 5, 37, 39]. Schizophrenia problems occur when the scope of a local declaration is left and re-entered during the same macrostep, so that one has to distinguish between the old and the new incarnations of local variables in a macrostep. Causality problems occur when the execution of a microstep depends on its result. As a consequence, causality analysis must be performed to detect and (if possible) to remove cyclic dependencies. Such dependencies become visible in the intermediate code, e.g., equation systems that are generated by the compiler.

Cyclic equation systems have been considered in detail in the analysis of combinational circuits with feedback loops in the early seventies [18, 19, 27]. Kautz [19] and Rivest [27] 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 [25, 26]. Furthermore, combinational cycles (called ‘false paths’ in this setting) occur in high-level synthesis of circuits by sharing common subexpressions [42]. As an example, consider the following equation (taken from [22]) where a system is to be implemented with large subsystems $f$ and $g$:

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

The cyclic equation system shown in Figure 1 requires only one instance of $f$ and one instance of $g$. In contrast, any acyclic version requires at least two instances of $f$ and $g$.

Malik [22] was the first who presented algorithms for the elimination of combinational cycles. Given input and output
variables $\vec{x}$ and $\vec{y}$, the problem is to check whether a Boolean equation system $\vec{y} = \Phi(\vec{x}, \vec{y})$ has a unique solution for all inputs. It is not difficult to see that this problem is NP-complete [22]. In order to obtain more efficient procedures, Malik used the embedding of Boolean algebra in ternary algebra as proposed by Bryant for the simulation of switch-level circuits [6]. 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 $f_\delta(\vec{y}) := \Phi(\vec{x}, \vec{y})$, where the inputs $\vec{x}$ are fixed. The existence of such fixpoints is guaranteed due to the embedding in the ternary domain by the Tarski-Knaster theorem [44] (see also the next section).

Malik’s procedure is able to compute an equivalent acyclic equation system $\vec{y} = \Phi(\vec{x})$. Although the acyclic version is known to require possibly more operations than the original cyclic one [18, 19, 27], this is a popular way for generating single-threaded sequential code [1] or, equivalently, a schedule [13] for the execution of the microsteps.

Recent work also considered alternatives to the fixpoint computation. In particular, [10, 24] reconsidered the causality problem as the theoretically more difficult satisfiability problem and proposed new SAT solving techniques and temporal induction for their solution.

Malik’s approach has been generalized by Shiple et al. [38, 39, 40] to sequential circuits. In this generalization, one considers equation systems of the following form, where $\vec{\omega}$ is a vector of Boolean constants, and $\Phi(\vec{x}, \vec{\ell}, \vec{y})$ and $\Omega(\vec{x}, \vec{\ell}, \vec{y})$ are vectors of Boolean expressions:

\[
\begin{align*}
\vec{y} &= \Phi(\vec{x}, \vec{\ell}, \vec{y}) \\
\text{init}(\vec{\ell}) &= \vec{\omega} \\
\text{next}(\vec{\ell}) &= \Omega(\vec{x}, \vec{\ell}, \vec{y})
\end{align*}
\]

This equation system contains input variables $\vec{x}$, output variables $\vec{y}$, and state variables $\vec{\ell}$. If the equation system is the result of the compilation of a synchronous program, then $\vec{y} = \Phi(\vec{x}, \vec{\ell}, \vec{y})$ represents the data flow, and the other equations represent the control flow of the program. It is easily seen that the control flow is acyclic, and hence deterministic, while the data flow may have cyclic dependencies.

Shiple’s analysis consists of two phases: In the first phase, Malik’s procedure is used to transform the data flow $\vec{y} = \Phi(\vec{x}, \vec{\ell}, \vec{y})$ into an equivalent acyclic version $\vec{y} = \Phi(\vec{x}, \vec{\ell})$. Note that this is always possible in the ternary domain, but not always in the Boolean domain. Hence, it may be the case that the right hand side evaluates to a non-Boolean value. For this reason, the second phase of Shiple’s analysis consists of checking whether a non-Boolean value can appear for one of the reachable states.

In the meantime, there exist alternative approaches for generating sequential code [11, 12]. However, these approaches still need causality analysis to guarantee the existence of a dynamic schedule for mutually dependent actions.

There are a lot of equivalent views on causality analysis: Shiple proved the equivalence to Brzozowski and Seger’s timing analysis in the up-bounded inertial delay model [8]. This means that a circuit derived from a cyclic equation system will finally stabilize for arbitrary gate delays iff the equation system is causally correct. Berry pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic and introduced the notion of constructive circuits [4]. The problem is also equivalent to type-checking in functional programs due to the Curry-Howard isomorphism [17]. Finally, Edwards reformulates the problem in that the existence of dynamic schedules must be guaranteed for the execution of mutually dependent microsteps [13]. Hence, causality analysis is a fundamental algorithm that has already found many applications in computer science.

Shiple’s method is currently used for the compilation of synchronous programs like Esterel programs [1, 12, 14] (although alternative heuristics are applied in a first instance [37]). However, the recent introduction of delayed actions as proposed in [30] (and now used in version 7 of Esterel-Studio [14]) requires to consider a more general problem. Delayed actions have been introduced to mimic sequential assignments so that algorithms given in single-threaded imperative languages can be easily implemented as synchronous programs. For example, the incrementation of a loop variable, i.e., the assignment $i := i + 1$, makes no sense in synchronous languages, since this requires to compute a solution to the (unsolvable) equation $i = i + 1$. Using delayed actions, one can write $\text{next}(i) := i + 1$, which is the intention of the assignment.

Moreover, delayed actions are convenient for describing hardware behavior. As an example, consider the implementation of a Russian multiplier in our Esterel variant called Quartz [30] given in Figure 2. Note that register transfers are easily modeled by means of delayed assignments. The result after compilation is depicted in Figure 3 where the states correspond to the locations (pause statements) of the

![Figure 1: An Example of a Cyclic Equation System](image1)

![Figure 2: Russian Multiplication in Quartz](image2)
The compilation of Quartz programs as presented in [30, 33] can be used to generate an equation system of the following form:

\[
\begin{align*}
\text{init}(\bar{y}) &= \Phi(\bar{x}, \bar{\ell}, \bar{y}) \\
\text{next}(\bar{y}) &= \Phi(\bar{x}, \bar{\ell}, \bar{y}, \text{next}(\bar{y})) \\
\text{init}(\bar{\ell}) &= \Omega \\
\text{next}(\bar{\ell}) &= \Omega(\bar{x}, \bar{\ell}, \bar{y})
\end{align*}
\]

In contrast to Shiple's equation systems, the data flow is split into initialization and transition parts that may both contain cyclic dependencies. Therefore, we say that the above equation system has a \textit{sequential data flow}. In the sequel, we focus on such equation systems and describe an algorithm to check causality of such equation systems. In case the causality is given, the equation system is transformed into an equivalent acyclic one that can finally be used for code generation.

The added value of this paper is therefore the extension of Malik's and Shiple's approaches to a causality analysis of systems with sequential data flow. This is necessary to compile synchronous programs with delayed actions. We show that similar to Shiple's extension, we can still rely on the fixpoint iteration used in [22]. However, we have to employ two fixpoint computations: one for the initialization part, and another one for the transition part. Similar to previous work, the fixpoints are computed in a lattice that extends the Boolean values by further elements \(\bot\) and \(\top\), so that their existence is guaranteed. For this reason, we have to check afterwards whether the fixpoints contain non-Boolean values. If they only contain Boolean values, we have transformed the equation system into an equivalent acyclic one and are therefore able to generate single-threaded code. Otherwise, it may be possible to generate single-threaded code, but an exact analysis is \textsc{np}-complete, so we conservatively decide to reject the program (this is typical for compilers of synchronous languages).

The outline of the paper is as follows: in the next section, we formulate the problem for combinational equation systems and review Malik’s procedure and its formal foundation with lattices and Tarski’s theorem. In Section 3, we present our extension of this procedure to equation systems with sequential data flow. In Section 4, we then discuss an implementation based on a dual rail encoding, where we directly encode the fact that inputs always take Boolean values. Section 5 illustrates our algorithm by a small example. In the appendix, we show that the analysis depends on the syntax of the formulas in the equation system, i.e., on the used Boolean operators, and even on the way they were extended to the non-Boolean values.

2. CYCLIC EQUATION SYSTEMS

In this section, we review the analysis of cyclic equation systems due to Malik [22]. His method is based on the fixpoint theorem of Tarski and Knaster [44] that can be applied after embedding the Boolean values \(\mathbb{B} = \{0, 1\}\) in a complete lattice \(\mathbb{P} = \{⊥, 0, 1, ∨\}\). With appropriate extensions of the Boolean operators, every equation system has fixpoints that can be used to construct an equivalent acyclic equation system.

2.1 Formulation of the Problem

The problem is to decide for a given equation system whether it defines \textit{unique outputs for all possible inputs}. In this section, we consider Boolean equation systems (hardware circuits) of the following form:

\[
\begin{align*}
y_1 &= \Phi_1(x_1, \ldots, x_m, y_1, \ldots, y_n) \\
&\vdots \\
y_n &= \Phi_n(x_1, \ldots, x_m, y_1, \ldots, y_n)
\end{align*}
\]

The propositional formulas \(\Phi_i(x_1, \ldots, x_m, y_1, \ldots, y_n)\) only depend on the variables \(x_i\) and \(y_i\). The variables \(x_1, \ldots, x_m\) 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 \(\bar{y} = \Phi(\bar{x}, \bar{y})\). The problem can be reduced to check the (unique) existence of fixpoints of the vector function \(f_\bar{x}(\bar{y}) = \Phi(\bar{x}, \bar{y})\). In particular, it is desirable to compute the fixpoint \(\bar{y} = \Phi(\bar{x})\) depending on the inputs \(\bar{x}\) in order to generate an equivalent acyclic equation system.

2.2 Fixpoint Theory

Fixpoint theory is well understood in theoretical computer science. In particular, the Tarski-Knaster theorem is often used to compute least and greatest fixpoints [44]. This theorem is fundamental to nearly all verification algorithms [32]. To apply this theorem and its related fixpoint iteration, we have to consider (complete) lattices.

A partially ordered set \((D, \sqsubseteq)\) is \textit{directed}, if all two-element sets \(\{x, y\} \subseteq D\) have upper and lower bounds in \(D\). A partially ordered set \((D, \sqsubseteq)\) is a \textit{lattice}, if all two-element sets \(\{x, y\} \subseteq D\) have suprema \(\sup\{x, y\}\) and infima \(\inf\{x, y\}\) in \(D\). Moreover, \((D, \sqsubseteq)\) is a \textit{complete lattice}, if for every set \(M \subseteq D\), \(\sup(M)\) and \(\inf(M)\) exist in \(D\). In particular, we write \(\bot := \inf(D)\) and \(\top := \sup(D)\) for the minimal and maximal element of \(D\).
A function \( f : D \to D \) is monotonic, if for all \( x, y \in D \) with \( x \sqsubseteq y \), we have \( f(x) \sqsubseteq f(y) \). Finally, \( f : D \to D \) is continuous, if \( f(\sup(M)) = \sup(f(M)) \) and \( f(\inf(M)) = \inf(f(M)) \) holds for all directed sets \( M \subseteq D \). It is easily seen that every continuous function is also monotonic [32]. However, the converse need not hold (not even in complete lattices). The theorem of Tarski and Knaster. For finite sets \( D \), it can be shown that every monotonic function is continuous. Moreover, every finite lattice is complete.

**Theorem 1 (Tarski/Knaster Theorem [44]).**

Let \( (D, \sqsubseteq) \) be a complete lattice and \( f : D \to D \) be a monotonic function. Then, \( f \) has fixpoints and the set of fixpoints even has a minimum \( \hat{x} \) and a maximum \( \hat{x} \). If \( f \) is moreover continuous, then the least fixpoint \( \hat{x} \) of \( f \) can be computed by the iteration \( p_0 := \bot, p_{n+1} := f(p_n) \), and the greatest fixpoint \( \hat{x} \) of \( f \) can be computed by the iteration \( q_0 := \top, q_{n+1} := f(q_n) \).

The above form is a general formulation of the Tarski-Knaster theorem. For finite sets \( D \), it can be shown that every monotonic function is continuous. Moreover, every finite lattice is complete.

### 2.3 Embedding Booleans in a Lattice

In order to apply the above theorem to causality analysis, we have to embed the Boolean values \( \{0, 1\} \) in a lattice so that the extensions of all Boolean functions are monotonic. We can achieve this by extending \( B = \{0, 1\} \) with the new elements \( \bot \) and \( \top \) to the set \( F = \{\bot, 0, 1, \top\} \). Moreover, we define the following partial order \( \sqsubseteq \) (of course, we mean its reflexive-transitive closure):

\[
0 \sqsubseteq \bot \sqsubseteq 1
\]

As usual, we can extend this partial order on \( F \) to a partial order relation on \( F^n \) so that the corresponding components of the vectors are compared. The next step is to extend every Boolean function \( f : B^n \to B^n \) to a function \( g_f : F^n \to F^n \) such that the following holds:

- \( \forall g \in B^n, g_f(\bar{g}) = f(\bar{g}) \)
- \( g_f : F^n \to F^n \) is monotonic

To this end, we have to extend the basic Boolean operators \( \neg, \land, \lor \) to corresponding operators \( \bar{\neg}, \bar{\land}, \bar{\lor} \) on \( F \). The monotonicity is a strong requirement that predetermines most of the extensions: In particular, there is only one possible extension \( \bar{\neg} \) for the negation \( \neg \).

To determine an extension \( \bar{\land} \) for \( \land \), first consider the possible values for \( 1, \bot \). Because of the desired monotonicity, it is required that \( \bot \land \bot = 0 \land 0 = 0 \) and that \( \bot \land 1 = 1 \land \bot = 1 \) holds. Hence, the only possibility is to define \( \bot \land 1 := \bot \). In the same way, we derive \( 1 \land \bot := 1 \). Because of \( \bot \land \bot = \bot \land 1 = \bot \), we moreover obtain \( 1 \land \bot := 1 \). Similar observations lead to the following (incomplete) operator tables:

\[
\begin{array}{ccccc}
\land & \bot & 0 & 1 & \top \\
\bot & \bot & 0 & 1 & \top \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\top & \top & 1 & 1 & 1 \\
\end{array}
\]

There are now different ways to proceed. It is reasonable to demand that \( 0 \land x = 0 \) holds, and that \( \top \land \top = \top \) should be commutative. For \( \land \) and \( \lor \), there is only one choice left, namely \( \top \land \top = \bot \land \bot = \bot \) and \( \top \land \bot = \bot \land \top = \bot \), which have to be defined as shown below. The reason for this is as follows (we only explain \( \top \land \bot \)). If \( f(M) = \bot \) implies \( \top \land \bot = \bot \), then \( \top \land \bot = \bot \) implies \( \top \land \bot = \bot \). Hence, we have to define \( \top \land \bot := 1 \).

This completes our definition of a four-valued logic \( F \). Although it is not the only possibility to extend \( B \), it is the best2 such extension for causality analysis, because the expressions evaluate as often as possible to Boolean values \( 0, 1 \). The following lemma lists some facts about the algebraic structure of \( F \):

**Lemma 1 (Algebraic Laws of \( F \)).** For conciseness, we abbreviate \( x \land y := \inf((x, y)) \) and \( x \lor y := \sup((x, y)) \) in the lattice \( F = \{\bot, 0, 1, \top\} \). Then, the functions \( \neg, \land, \lor, \sup, \inf \) are monotonic in all arguments, and the functions \( \sup, \inf \) have fixpoints. Moreover, the equations given in Figure 4 are valid. The restriction to \( B = \{0, 1\} \) is a Boolean algebra \( (B, 0, 1, \land, \lor, \neg) \), and the restriction to \( T = \{0, 1, \top\} \) is a ternary algebra \( (T, 0, 1, \bot, \top, \land, \lor, \neg, \sup, \inf) \).

However, we do not have the complement laws \( \neg \neg \neg x = 0 \) and \( \neg \neg \neg x = 1 \). Therefore, neither \( \neg, \sup, \inf \) nor \( (F, \bot, \top, \land, \lor, \neg) \) is a Boolean algebra. Only the restriction to \( B \) makes these laws valid. In a similar way, we do not have \( (\sup \sup x) \sup = x \sup \sup x \sup \) which is required in ternary algebra [8], so that only the restriction to \( T \) makes this equation valid.

2However, there are good reasons why one might not be interested in finding arbitrary Boolean solutions. One might argue that the way they have been computed is not natural. (See the discussion in the appendix for more details.)
2.4 Causality Analysis by Fixpoint Iteration

As monotonic functions are closed under function composition, all extensions \( g_f : \mathbb{B}^n \to \mathbb{B}^n \) of function \( f : \mathbb{B}^n \to \mathbb{B}^n \) obtained by replacing \( \land, \lor \) and \( \neg \) with \( \land, \lor \) and \( \lor, \land \), respectively, are monotonic. Hence, it follows from the Tarski-Knaster theorem, that these functions have fixpoints in \( \mathbb{B}^n \), and even more, they have a least and a greatest fixpoint (that may be identical). The Tarski-Knaster theorem also states how these fixpoints can be computed: we use the iteration \( y_{i+1} := \Phi^i(x, y_i) \) that has to be started with \( y_0 := (\bot, \ldots, \bot) \) for computing the least fixpoint and with \( y_0 := (\top, \ldots, \top) \) for computing the greatest fixpoint.

If the least fixpoint \((y_1, \ldots, y_n)\) belongs to \( \mathbb{B}^n \), it follows that the equation system has a unique Boolean solution. Moreover, it follows that the equation system has a unique Boolean solution: if there would be another fixpoint \((\bar{y}_1, \ldots, \bar{y}_n)\), then it follows by the minimality of \((y_1, \ldots, y_n)\) that we must have \( \bar{y}_i \subseteq y_i \) for all \( i \in \{1, \ldots, n\} \). However, the Boolean values 0 and 1 are incomparable in \( \mathbb{B} \), so that we conclude that either \( \bar{y}_i = y_i \) or \( y_i = \top \) must hold. Thus, there is no other Boolean solution in this case.

**Lemma 2 (Fixpoints in \( \mathbb{B}^n \)).** Given a Boolean vector function \( \bar{f} : \mathbb{B}^n \to \mathbb{B}^n \) and the least (\( y_1, \ldots, y_n \)) and greatest fixpoints \((\bar{y}_1, \ldots, \bar{y}_n)\) of the extended function \( \bar{g} : \mathbb{B}^n \to \mathbb{B}^n \), the following holds for \( i \in \{1, \ldots, n\} : \)

- \( y_i \in \mathbb{B} \) implies \( \bar{y}_i = y_i \)
- \( \bar{y}_i \in \mathbb{B} \) implies \( \bar{y}_i = y_i \)
- \( \bar{y}_i = y_i \) implies \( \bar{y}_i \in \mathbb{B} \)
- \( \bar{y}_i = y_i \) implies \( \bar{y}_i = \top \) and \( y_i = \top \)

The proof is based on the fact that the computation of the least and the greatest fixpoints are dual in that \( \bot \) and \( \top \) are exchanged. According to the operation tables, it therefore follows that \( \top \) can never occur in the computation of least fixpoints, and \( \bot \) can never occur in the computation of greatest fixpoints. Hence, these computations are performed in substructures of \( \mathbb{B}^n \) that are ternary algebras [8]. For this reason, it is sufficient to compute either the least or the greatest fixpoint for causality analysis: Either both fixpoints are the same, or none of them belongs to \( \mathbb{B}^n \).

**Theorem 2 (Causality Analysis).** Given a Boolean vector function \( \bar{f} : \mathbb{B}^n \to \mathbb{B}^n \) and the least (greatest) fixpoint \( \bar{y} (\bar{y}) \) of the extended function \( \bar{g} : \mathbb{B}^n \to \mathbb{B}^n \), the following is equivalent:

- \( \bar{y} = \bar{y} \)
- \( \bar{y} \in \mathbb{B}^n \)
- \( \bar{y} \in \mathbb{B}^n \)
- \( \bar{y} = \bar{f}(\bar{y}) \) has only one solution in \( \mathbb{B}^n \)

However, if \( \bar{y} \in \mathbb{B}^n \) holds, it follows that \( \bar{y} \neq \bar{y} \) and \( \bar{y} \notin \mathbb{B}^n \). In this case, we know nothing about the existence of Boolean solutions of \( \bar{y} = \bar{f}(\bar{y}) \).

In general, equation systems may have more than one solution in the four-valued domain. In particular, it may be the case that neither the least \( \bar{y} \) nor the greatest fixpoint \( \bar{y} \) belongs to \( \mathbb{B}^n \), even if unique Boolean solutions \( \bar{y} \) exist for all inputs. Synchronous programs of that type are called logically correct, but not constructive. Although they have unique Boolean solutions, we are not able to find this Boolean solution with the above fixpoint computation. While this is certainly a drawback of the procedure, the advantages are predominant: instead of checking satisfiability of the equation system which is NP-complete, we can compute the fixpoints with at most \( 2n \) iterations (this is the diameter of \( \mathbb{B}^n \), i.e., the length of the largest chain in \( \mathbb{B}^n \)), and each iteration can be computed in linear time with respect to the size of the equation system.

3. Analysis with Delayed Actions

In the previous section, we described how Tarski’s fixpoint iteration is used to find solutions of Boolean equation systems. In this section, we generalize this procedure to handle equation systems that stem from synchronous programs with delayed actions [30].

Boolean equation systems as considered in the previous section can only be used to reason about combinatorial behavior. To analyze sequential systems, Shipley et. al. [38, 39, 40] already generalized Malik’s method for equation systems of the following form with \( \bar{w} \in \mathbb{B}^n \):

\[
\begin{align*}
\bar{y} &= \bar{f}(x, \bar{e}, \bar{y}) \\
\text{init}(\bar{f}) &= \bar{w} \\
\text{next}(\bar{f}) &= \check{\bar{f}}(x, \bar{e}, \bar{y})
\end{align*}
\]

Again, \( x = (x_1, \ldots, x_m) \), \( \bar{e} = (e_1, \ldots, e_q) \), and \( \bar{y} = (y_1, \ldots, y_n) \) are Boolean valued vectors where \( x_1, \ldots, x_m \) are input variables, \( y_1, \ldots, y_n \) are output variables, and \( e_1, \ldots, e_q \) are state variables. The equation \( \bar{y} = \bar{f}(x, \bar{e}, \bar{y}) \) describes the data flow, whereas \( \text{init}(\bar{f}) = \bar{w} \) and \( \text{next}(\bar{f}) = \check{\bar{f}}(x, \bar{e}, \bar{y}) \) describe the control flow of the program.

It is important to understand when the equations must hold. The data flow equations in \( \bar{y} = \bar{f}(x, \bar{e}, \bar{y}) \) impose invariants and must hold at every point of time. In contrast, the control flow is given recursively: the initial equation \( \text{init}(\bar{f}) = \bar{w} \) determines the values of the state variables \( \bar{e} \) at the initial point of time, which means that the equation system \( \bar{e} = \bar{w} \) must hold at \( t = 0 \). The values of the state variables for \( t > 0 \) are determined by the transition equations \( \text{next}(\bar{f}) = \check{\bar{f}}(x, \bar{e}, \bar{y}) \) that deterministically compute the next state \( \text{next}(\bar{f}) \) from the current inputs \( x \), the current state \( \bar{e} \), and the current outputs \( \bar{y} \).

Equation systems as the one above are sufficient to describe the semantics of synchronous programs with immediate actions. However, the use of delayed actions results in more general equation systems of the following form:

\[
\begin{align*}
\text{init}(\bar{g}) &= \Psi(x, \bar{e}, \bar{g}) \\
\text{next}(\bar{g}) &= \Psi(x, \bar{e}, \bar{g}, \text{next}(\bar{g})) \\
\text{init}(\check{\bar{g}}) &= \bar{w} \\
\text{next}(\check{\bar{g}}) &= \check{\bar{f}}(x, \bar{e}, \bar{g})
\end{align*}
\]

In such equation systems, cyclic dependencies might occur at two places: firstly, in the initialization of the data flow, i.e., in the equation system \( \text{init}(\bar{g}) = \Psi(x, \bar{e}, \bar{g}) \), and secondly, in the recursion of the data flow, i.e., in the equation system \( \text{next}(\bar{g}) = \Psi(x, \bar{e}, \bar{g}, \text{next}(\bar{g})) \). Hence, we have to solve two fixpoint problems that are related with the data flow at time \( t = 0 \), and at time \( t > 0 \). To this end, we embed our equation system in the lattice \( \mathbb{L}^n \) and compute the solutions of the following equation systems as outlined in the previous section:
We may additionally assume that the inputs $\vec{x}$ as well as the current state variables $\vec{\ell}$ take Boolean values. Moreover, note that we neglect the temporal relationship between current outputs $\vec{y}$ and outputs at the next point of time $\vec{y}'$ (or next($\vec{y}$)) in that we view these as different variables when the fixpoints are computed. The reason for this is that the iteration steps of the fixpoint computation correspond to microsteps of the program, and therefore to the same point of time.  

In the next section, we will see how the fixpoints can be computed symbolically, i.e., depending on the input and state variables always have Boolean values. Moreover, we neglect the temporal relationship between current outputs $\vec{y}$ and outputs at the next point of time $\vec{y}'$ (or next($\vec{y}$)) in that we view these as different variables when the fixpoints are computed. The reason for this is that the iteration steps of the fixpoint computation correspond to microsteps of the program, and therefore to the same point of time.

In the next section, we will see how the fixpoints can be computed symbolically, i.e., depending on the input and state variables. The results are then two equivalent acyclic equation systems, say $\vec{y} = \vec{\Phi}(\vec{x})$ and $\vec{y}' = \vec{\Phi}(\vec{x}, \vec{\ell}, \vec{y}, \vec{y}')$. It may now be the case that either $\vec{\Phi}(\vec{x})$ or $\vec{\Phi}(\vec{x}, \vec{\ell}, \vec{y}, \vec{y}')$ evaluates to a non-Boolean vector for certain Boolean values of $\vec{x}, \vec{\ell}$, and $\vec{y}$. However, these bad values may never occur, because the corresponding states are not reachable. For this reason, a reachability analysis has to check if such assignments can occur.

To this end, we combine the acyclic equation systems with the control flow:

\[
\begin{align*}
\text{init}(\vec{y}) & = \vec{\Phi}(\vec{x}) \\
\text{next}(\vec{y}) & = \vec{\Phi}(\vec{x}, \vec{\ell}, \vec{y}, \vec{y}') \\
\text{init}(\vec{\ell}) & = \vec{\omega} \\
\text{next}(\vec{\ell}) & = \vec{\Pi}(\vec{x}, \vec{\ell}, \vec{y})
\end{align*}
\]

Note that we now take care of the fact that $\vec{y}'$ is the value of $\vec{y}$ at the next point of time. A reachability analysis has to check if we can reach a state where $\vec{y}'$ takes a non-Boolean value. We can thereby assume that the inputs as well as the state variables always have Boolean values, since a state variable may only evaluate to a non-Boolean value if a non-Boolean value of an output at the previous point of time propagates through. In the next section, we will explain how this knowledge is incorporated in the analysis.

### 4. IMPLEMENTATION

In the previous sections, we have defined the four-valued lattice $\mathbb{P}$ with monotonic extensions $\Rightarrow$, $\Rightarrow$, and $\check{\Rightarrow}$ of the Boolean operators $\Rightarrow$, $\wedge$, and $\lor$. The causality analysis requires to compute fixpoints for particular inputs and particular states. In this section, we show how this can be done symbolically, i.e., by considering all inputs and all states at once in a single fixpoint iteration.

The method used here has been proposed by Bryant for ternary simulation of MOS transistor circuits [6]. However, we directly encode the fact that input and state variables are always Boolean values. This fact can be simply encoded, which saves a lot of propositional variables and therefore makes our analysis more efficient.

#### 4.1 Dual Rail Encoding

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

**DEFINITION 1 (DUAL-RAIL ENCODING).** For the encoding of the values of $\mathbb{P}$ by values of $\mathbb{B}^2$, we use the following bijective mapping $\epsilon : \mathbb{B}^2 \rightarrow \mathbb{P}$:

<table>
<thead>
<tr>
<th>$x \in \mathbb{B}^2$</th>
<th>$\epsilon(x) \in \mathbb{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 0)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$(0, 1)$</td>
<td>$0$</td>
</tr>
<tr>
<td>$(1, 0)$</td>
<td>$1$</td>
</tr>
<tr>
<td>$(1, 1)$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>

We moreover define the following operations on $\mathbb{B}^2$:

- $\neg_2(x_1, x_2) := (x_2, x_1)$
- $(x_1, x_2) \wedge_2 (y_1, y_2) := (x_1 \wedge y_1, x_2 \vee y_2)$
- $(x_1, x_2) \vee_2 (y_1, y_2) := (x_1 \vee y_1, x_2 \wedge y_2)$
- $(x_1, x_2) \leq_2 (y_1, y_2) := (\neg x_1 \vee y_1) \land (\neg x_2 \vee y_2)$
- $\inf((x_1, x_2), (y_1, y_2)) := (x_1 \land y_1, x_2 \wedge y_2)$
- $\sup((x_1, x_2), (y_1, y_2)) := (x_1 \lor y_1, x_2 \vee y_2)$

Of course, the above definition has been made with some care. In particular, it turns out that the above operations on $\mathbb{B}^2$ are related to the corresponding ones on $\mathbb{P}$:

**THEOREM 3 (DUAL-RAIL ENCODING).** The functions $\neg_2$, $\wedge_2$, $\vee_2$, $\leq_2$, $\inf$, and $\sup$ are the homomorphic versions of $\neg$, $\wedge$, $\lor$, $\leq$, $\inf$, and $\sup$ on $\mathbb{P}$, i.e., for all $x, y \in \mathbb{B}^2$ we have:

- $\epsilon(\neg_2 x) = \neg(x)$
- $\epsilon(x \wedge_2 y) = \epsilon(x) \wedge \epsilon(y)$
- $\epsilon(x \vee_2 y) = \epsilon(x) \vee \epsilon(y)$
- $\epsilon(x \leq_2 y) :\iff \epsilon(x) \leq \epsilon(y)$
- $\epsilon(\inf(x, y)) = \inf\{\epsilon(x), \epsilon(y)\}$
- $\epsilon(\sup(x, y)) = \sup\{\epsilon(x), \epsilon(y)\}$

### 4.2 A Semantic Interpretation

Besides the fact that dual rail encoding reduces all problems concerning functions on $\mathbb{P}$ to corresponding Boolean problems, there is also a nice interpretation of this encoding; Berry [3] describes the constructive semantics of Esterel in that he defines two sets of signals $\text{must}_e(S)$ and $\text{can}_e(S)$ for every statement $S$ and every ternary variable assignment $g : V \rightarrow \{0, 1, \bot\}$. The idea is thereby that these two sets are conservative approximations of the data flow: for the variable assignment $g : V \rightarrow \{0, 1, \bot\}$, $\text{must}_e(S)$ computes those variables that must be instantaneously emitted, and $\text{can}_e(S)$ computes those variables that can be instantaneously emitted. By the definition given in [3], it can be easily proved that $\text{must}_e(S) \subseteq \text{can}_e(S)$ holds.

Given a translation from synchronous programs to Boolean equation systems, there is a close relationship between the dual rail encoding and the $\text{must}_e(S)$ and $\text{can}_e(S)$ approximations: To see this, note first that we can represent the sets of signals $\text{must}_e(S)$ and $\text{can}_e(S)$ by Boolean valued functions $\text{must}_e(S)$ and $\text{can}_e(S)$, respectively, that represent the characteristic functions of these sets. Then, the dual rail encoding consists of the pair $\langle \text{must}_e(S), V \setminus \text{can}_e(S) \rangle$.

This view of having optimistic and pessimistic approximations also holds for the analysis of Boolean equation systems that we have considered so far in this paper. To see this, consider how the fixpoint iteration described before computes the outputs $\vec{y}$ of an equation system $\vec{y} = \vec{\Phi}(\vec{x}, \vec{y})$ for a given
of input variables $x$. To this end, assume that the current assignment is given by a function $\varrho : \mathcal{V} \rightarrow \{0, 1\}$ that assigns either 1 or 0 to input variables $x_k$, and ⊥ to all output variables $y_k$. We then compute new assignments $g_1$, $g_2$, $g_3$ by the fixpoint iteration described before. Now consider how the value of an output variable $y_k$ is updated during this iteration: Given that the result of the evaluation of the right hand side $\Phi_k(x, \bar{y})$ is $(v_1, v_2)$, we define:

$$g_{i+1}(y_k) := \begin{cases} 
\bot & \text{if } (v_1, v_2) \equiv (0, 0) \\
0 & \text{if } (v_1, v_2) \equiv (0, 1) \\
1 & \text{if } (v_1, v_2) \equiv (1, 0) 
\end{cases}$$

Obviously, this is just the dual rail encoding. The interpretation offered by Berry [3] is as follows: if $v_1$ holds, the signal must be true, if $v_2$ holds, the signal can not be true. This matches exactly the encoding we have used above:

$$(v_1, v_2) \equiv (0, 0):$$ Since $v_1 \equiv 0$ holds, we can not argue that the variable must be true. Since $v_2 \equiv 1$ holds, we can not argue that the variable can not be true. Hence, we can say nothing about the value at this iteration, and retain the value ⊥ for the next iteration.

$$(v_1, v_2) \equiv (0, 1):$$ Since $v_2 \equiv 1$ holds, we conclude that the variable can not be true. Hence, the value of $y_k$ is now determined to the ternary 0.

$$(v_1, v_2) \equiv (1, 0):$$ Since $v_1 \equiv 1$ holds, we conclude that the variable must be true. Hence, the value of $y_k$ is now determined to the ternary 1.

$$(v_1, v_2) \equiv (1, 1):$$ Due to our operation tables, this value can never occur, when we start with an assignment that does never map variables to ⊤.

In any case, the next value $g_{i+1}(y_k)$ is directly encoded as the result of the evaluation $(v_1, v_2)$. Hence, the dual rail encoding is a well chosen encoding such that the first component is a pessimistic approximation (it only assigns 1 if this can not be avoided), and the second one is the complement of the most optimistic approximation (it assigns 1 if there is no reason why not to assign this value). This view easily proves the equivalence of the analysis given by the must, (S) and can, (S) definitions given in [3]. We discuss variants of this analysis in the appendix.

4.3 Symbolic Implementation by BDDs

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. Thus, canonical normal forms for propositional logic like binary decision diagrams (BDDs) [7, 41] lead to canonical normal forms for functions on $\mathbb{F}$. To obtain an efficient implementation, we can therefore easily use BDDs for representing and manipulating these formulas.

To this end, it is advantageous to exploit the fact that input and state variables are always Booleans. Figure 5 shows the function Rails that computes for a given ‘propositional’ formula $\Phi$ and a set of variables $\mathcal{Y}$ two propositional formulas $(\varphi, \psi)$ that are the two rails of $\Phi$, i.e., the formulas that encode the corresponding function on $\mathbb{B}^\mathcal{Y}$. For notational convenience, we encode a variable $x \in \mathcal{Y}$ over $\mathbb{F}$ by the two Boolean variables $(x.r1, x.r2)$ denoting the two rails of $x$.

Note that for variables $x \not\in \mathcal{Y}$, we return the pair $(x, \neg x)$ which means that we have 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 can save a lot of variables to minimize the BDD sizes.

![Figure 5: Computing the Rails of a Formula](image)

![Figure 6: Causality Analysis with BDDs](image)
always complementary. In this case, the outputs are Booleans and we can use their first rails for code generation.

Table 1: A Cyclic Quartz Program

5. EXAMPLE

To illustrate the algorithms of the previous section, consider the Quartz program given in Figure 7. According to the compilation technique presented in [30, 33], we obtain the equation system given in Figure 8.

As can be seen, there is a cyclic dependency of \(\text{next}(y_2)\). The initialization equations, however, are acyclic, so that there is no need to analyze the initialization phase. To analyze the transitions, we consider the following part of the equation system of Figure 8:

Let us now interpret the above equation system over \(\mathbb{F}_2\) and encode the problem in \(B^3\) with dual rail encoding:

We can now compute the least fixpoint starting with values \(\text{next}(y_0) = 1\), i.e., with the rails \((\text{next}(y_0), r_1, \text{next}(y_0), r_2) = (0, 0)\). For our example, the fixpoint iteration terminates after one iteration, i.e., the next iteration yields the same Boolean functions:

Figure 9: Acyclic Equation System for Program \(Q_1\)

We obtain the acyclic equation system given in Figure 9. It remains to check if the outputs in this equation system may take non-Boolean values. To this end, we employ a reachability analysis to check if the following formula invariantly holds (it states that all outputs \(y_i\) are Booleans):

\[(y_0, r_1 \oplus y_0, r_2) \land (y_1, r_1 \oplus y_1, r_2) \land (y_2, r_1 \oplus y_2, r_2)\]

A model checker like SMV [23, 9] can easily verify that this condition holds, since \(0 \lor \ell_1\) always holds. Hence, all outputs are Booleans, and therefore their first rails carry the values we are interested in.

The final step is to generate single-threaded code. To this end, we neglect the second rails, since (1) we know that they are dual to the first rails, and (2) according to our encoding the first rail is the desired Boolean value. We then obtain the equivalent acyclic equation system given in Figure 10. It is now straightforward to generate sequential code of this equation system by simply writing a function that evaluates these equations for given inputs and states.

6. CONCLUSIONS

In this paper, we considered a fundamental problem that appears in the compilation of synchronous programs, namely the analysis and elimination of cyclic dependencies. In contrast to previous work that is restricted to systems with a combinational data flow, our approach can deal with equation systems that contain a sequential data flow. Such equation systems are obtained by compiling synchronous programs with delayed actions. To this end, we extended classical approaches by applying fixpoint-based analysis techniques to the data flow.
init(y₀) = 1
init(y₁) = 0
init(y₂) = 0
next(y₀) = (ℓ₀ ∨ ℓ₁) ∧ ¬y₀
next(y₁) = (ℓ₀ ∨ ℓ₁) ∧ ¬y₁
next(y₂) = 0
init(ℓ₀) = 1
init(ℓ₁) = 0
next(ℓ₀) = 0
next(ℓ₁) = ℓ₀ ∨ ℓ₁

Figure 10: Final acyclic equation system for program Q₁

The main purpose of our method is to transform an equation system with cyclic dependencies into an equivalent acyclic one. Thus, it can be used for the translation of Quartz or Esterel programs with delayed actions to acyclic equation systems. Given such an acyclic equation system, it is straightforward to generate single-threaded code that can either be used for simulation or synthesis (both software and hardware). As a major benefit, programs can be run on a single processor without process management by an operating system.

7. REFERENCES

APPENDIX

A. TERNARY CONDITIONALS

We have embedded the Boolean operators $\neg$, $\land$, and $\lor$ in a four-valued lattice $\mathbb{F} = \{1, 0, 1', 1''\}$ such that the extended operators $\neg\!, \land\!, \lor\!$ are monotonic w.r.t. the ordering $\sqsubseteq$ on $\mathbb{F}$. Interpreted on the Boolean values, it does not make a difference if equivalences are applied to reduce $\neg\!, \land\!$, and $\lor\!$ to other Boolean operators. However, such modifications influence the result of the fixpoint computation. Hence, causality is a syntactical property: As we will see below, there are logically equivalent Boolean equation systems $E_{12}$ and $E_{12}'$, where $E_{12}'$ is causally correct, but $E_{12}$ is not.

To show these differences, consider the ‘if-then-else’ operator which is defined as follows:

$$\land\! \equiv (\alpha \Rightarrow \beta | \gamma) : \equiv \alpha \land \beta \lor \neg\! \alpha \land \gamma.$$  

This operator is used for the compilation of synchronous programs at several stages. For example, the definition of the condition whether a statement is executed in zero-time, i.e., whether it is instantaneous could be defined as follows (see [30] for a complete definition):

- \text{instant \( (if \ \sigma \ then \ S_1 \ else \ S_2 \ end) \equiv (\sigma \Rightarrow \text{Act}(S_1) \lor \text{Act}(S_2)) \)}

Moreover, the set of signals that are emitted at the current macrostep are defined as follows:

- \text{Act \( (if \ \sigma \ then \ S_1 \ else \ S_2 \ end) \equiv (\sigma \Rightarrow \text{Act}(S_1) \lor \text{Act}(S_2)) \)}

The compilation of conditionals and sequences often affects the causality of a program. In particular, it has to be defined what has to be done in case the condition of a conditional statement evaluates to a non-Boolean value. These decisions can be implemented by different choices for the embedding of the if-then-else operator in the four-valued lattice $\mathbb{F}$.

The results are different when we retain the condition operators instead of replacing them by $\neg\!, \land\!, \lor\!$. To this end, we have to proceed as for the other operators, i.e., we have to define a function $f : \mathbb{F} \rightarrow \mathbb{F}$ such that (1) its restriction to $\mathbb{B}^3$ is the if-then-else operator, i.e., that $f(\alpha, \beta, \gamma) = (\alpha \Rightarrow \beta | \gamma)$, and (2) that $f$ is monotonic. Again, the predetermined Boolean cases and the required contraints due to monotonicity already determine many values. There are only few cases left where we can choose between a Boolean and a non-Boolean value. As we want the fixpoints to evaluate to Boolean values as often as possible, we prefer again the Boolean values. Therefore, we end up with the operator tables given in the lower part of Figure 11.

As can be seen, the two possible embeddings of $\land\!$ are not equivalent. Even more, none of them is superior, since there are places where one has a Boolean value while the other has a non-Boolean value. Looking carefully at the operator tables reveals the different views taken by the two embeddings: In the definition given in the lower part of Figure 11, we select $\beta$ if $\beta = \gamma$ holds, regardless of the value of
The causality condition \( \ell_0 \oplus -\ell_0 \) is valid, so that we have computed the following acyclic equation system:

\[
\begin{align*}
\{ \ y &= \ell_0 \\
next(\ell_0) &= 0 
\end{align*}
\]

Hence, we could say that the second variant is better, since it is able to compile the program. However, there are reasons against this choice: In \( P_{12} \), the information flows backwards: Reaching the conditional, the value of \( y \) is not yet determined, and can only be determined by looking ahead.

For this reason, we see that deciding causality at the level of Boolean equation systems depends on the syntax of the generated formulas in the equation system. In particular, simple logic optimizations like \( x \lor \neg x = 1 \) or \( x \lor \neg y = \neg y \lor \neg y \) have an essential impact on the causality.

In general, it has to be discussed what the aim of the causality analysis should be. In this paper, we have the view that causality analysis should check whether an equation system has a unique solution. This problem is known to be NP-complete, and therefore, we are interested in good heuristics. The causality analysis presented here can be seen as such a heuristic. As our aim is to find Boolean solutions as often as possible, we preferred Boolean values in the definition of the extensions of the operators. As a consequence, we chose the lower part of Figure 11. Thus, we have \( (\alpha \Rightarrow \beta \gamma) = \beta \), even if \( \alpha \) is not a Boolean value. In contrast, the current view taken by the Esterel compiler is that programs like \( P_{12} \) should not be regarded as correct programs, since such programs are not ‘well-formed’, because information flows backwards.

For this reason, one would rather choose an extension such that \( (\uparrow \Rightarrow \beta \gamma) = \bot \) holds.

To sum up, we obtain different versions of causality if we embed the Boolean operators like the if-then-else operator differently, or if we apply logic simplifications on the generated equation systems. At the program level, this has been pointed out by Boussinot in [5]. He presented several variants of \( \text{must}_y(S) \) and \( \text{can}_y(S) \) with different notions of causality. We tried several examples, in particular those listed in [5], and found that many equation systems can be made causally correct by applying the absorption law \( x \lor y = x \lor \neg x \land y \) and other simple Boolean laws before causality analysis.