Bachelor's thesis

Causal Correctness as a Safety Property

Matthias Lederer

2019-05-02

Examiner: Prof. Dr. Klaus Schneider
Supervisor: M.Eng. Xiao Wang

Abstract

Causality is one of the main obstacles in the compilation of synchronous programs with instantaneous cycles. We present a formulation of causal correctness/constructiveness of a synchronous program as a safety property on a synchronous circuit. This allows us to leverage advanced model checking procedures for verifying causal correctness of general boolean synchronous programs with least fixpoint semantics.

We implemented our method for programs written in Quartz represented as a set of guarded actions. The guarded actions are first translated into a synchronous circuit with cyclic combinational part where the result of one cycle is defined as a least fixpoint. Several methods are then presented for transforming the cyclic circuit into an equivalent acyclic one. Finally, we compare the performance of the different approaches using a model checker based on Property Directed Reachability.

Zusammenfassung

Kausalität ist eines der größten Hindernisse für die Compilierung von synchronen Programmen mit verzögerungsfreien Zyklen. In dieser Arbeit wird die kausale Korrektheit bzw. Konstruktivität eines synchronen Programms als eine Bad-State Property über einem synchronen Schaltwerk formuliert. Dies ermöglicht den Einsatz fortgeschrittener Model-Checking-Methoden für die Verifizierung kausaler Korrektheit von allgemeinen boolschen synchronen Programmen, deren Semantik über einen kleinsten Fixpunkt definiert ist.

Implementiert haben wir unsere Methode für Programme, die in der Sprache Quartz geschrieben wurden und als eine Menge bedingter Aktionen übergeben werden. Die bedingten Aktionen werden zunächst in ein synchrones Schaltwerk übersetzt, dessen Übergangsfunktion als der kleinste Fixpunkt auf einem zyklischen Schaltkreis gegeben ist. Anschließend werden mehrere Möglichkeiten dargestellt, um den zyklischen Schaltkreis in einen äquivalenten azyklischen zu überführen. Schließlich wird die Performanz der verschiedenen Ansätze bezüglich eines Modellprüfers basierend auf Property Directed Reachability verglichen.

Contents

1. Introduction **4**

2. Related Work **7**

3. Quartz **8**
- 3.1. Overview . . . 8
- 3.2. Quaternary Logic . . . 9
- 3.3. Guarded Actions . . . 10
- 3.4. Equation System from Guarded Actions . . . 12

4. Acyclic Circuit from Equation System **14**
- 4.1. Unrolling . . . 14
- 4.2. Fixpoint in Steps . . . 18

5. Evaluation **19**
- 5.1. Implementation . . . 19
- 5.2. Comparison . . . 19
- 5.3. Test Cases . . . 20
  - 5.3.1. Rivest . . . 20
  - 5.3.2. Arbiter . . . 22
  - 5.3.3. AsyncArbiterBoch82 . . . 23

6. Conclusion & Outlook **25**

Bibliography **27**

A. Time Complexity **29**

1. Introduction

Since their inception in the 1980's, synchronous languages have proved a solid foundation for the development of safety-critical embedded software. Despite the difficulties of their compilation, they have some undeniable strengths which makes them a good choice for highly reactive/concurrent systems: While they provide well-defined notions of (deterministic) parallelism, control and dataflow aspects can be formulated concisely, even allowing graphical representations as statecharts or block diagrams. Still, due to their simplicity, semantics can usually be given in operational or denotational form and they are generally amenable to formal verification. [2]

The underlying principle is that computation happens instantaneously at the ticks of a single (basic) clock, i.e. all variables change values instantaneously so their values are fixed between ticks. Thus, synchronous languages basically describe (not necessarily finite) state machines of Mealy type, i.e. outputs depend on the state and the current inputs. Their distinctive feature however is that they provide a natural way to compose such machines/processes in parallel. Parallel composition is straightforward when there are no mutual dependencies between the processes. But when processes are allowed to communicate with each other, i.e. when the output of process A is the input to process B whose output is input to A again, how should their combined output be defined? This directly raises the question of causality: as both processes' outputs depend each other's output, it is not clear a priori *why* any output should be generated at all. We will see in this paper how this question can be answered efficiently. Before we can give a precise definition of what we consider a causally correct program, we first have to describe how computation is carried out in the synchronous languages.

Although the full details of their syntax are quite involved (e.g. there is the problem of schizophrenia), the imperative languages Esterel [3] and Quartz [15] allow a much simpler alternative description of programs as sets of guarded actions. A guarded action in its basic form is a statement $\chi \Rightarrow x = \alpha$ where χ is a propositional boolean formula over all variables, i.e. input, state and reaction, called the guard; x is a variable and α a (not necessarily boolean) expression over all variables. The next state of a program given as a set of guarded actions is determined as follows: At first, only the input variables are known. The values of the other variables are then inferred as follows: Given a guarded action $\chi \Rightarrow x = \alpha$ where χ is evaluates to true, we can set x to α when α is known; had x be known to already have a different value, we would have a write conflict. When the guards of all actions on x are known to be false, x is set to some default value through the so-called "reaction to absence". This process is continued until no progress can be made anymore or a write conflict occurs. As we will see in section 3.4, the computation of the reaction can naturally be defined by a least fixpoint. With

guarded actions, the parallel composition of two processes can now simply be defined as the union of their guarded actions.

A program is called constructive or causally correct if, starting from the inputs, all other values can be inferred and no write conflict occurs in all reachable states. Note that Esterel programs for instance cannot have write conflicts; as we will be considering Quartz programs however, we need this extended definition of constructiveness. It is clear, that the property of constructiveness is not modular: adding a process might introduce instantaneous loops that result in a deadlock. Besides that, reachability cannot be disregarded as it is of paramount importance e.g. for mutual exclusion. Unfortunately, checking causality is as a result a hard problem in general (PSPACE-complete for boolean programs, see appendix A; clearly undecidable with unbounded arithmetic).

In general, causality is less prominent in synchronous languages adhering to the dataflow paradigm. Lustre e.g. forbids instantaneous cycles completely so there is no causality issue. Going even further, one could demand that no input of a node should depend instantaneously on an output of that node. Then nodes are truly modular and can be compiled separately [10]. Still, there are sensible examples with instantaneous cycles (see e.g. section 5.3.2) and formulating them in Quartz or Esterel is not hard. Furthermore, the resulting circuits with cyclic combinational part could possibly be smaller than equivalent acyclic circuits and result in more power efficient systems [13]. Signal on the other hand is multiclock, i.e. flows do not need to have a value in every step, and the equations that make up the program are seen as constraints: the program does not necessarily define a unique reaction or any reaction at all. In the presence of multiple clocks however, the problem of clock consistency arises: some nodes like addition that are operating on multiple flows require that their inputs follow the same clock. Note that, in principle, slower clocks could be simulated using additional signals; clock consistency could then be formulated as a safety property as well and thus be verified with the same methods as used for verifying causal correctness.

As said before, Quartz will serve as the basis of discussion throughout this paper. However, we will not be treating programs on the syntactical level but will analyse causality on an equation system that we derive from the guarded actions. Even though modifications will be necessary, our method thus should apply to other synchronous languages as well, as long as the result of one cycle is defined as a least fixpoint. As it stands, our analysis is limited to checking causalitiy of boolean programs only, i.e. programs that operate on more complex datatypes will have to be converted so that they operate on single bits. Implementing operations on those datatypes in boolean logic should not cause too much of a blowup as it is required for logic synthesis anyway.

While most approaches for checking causal correctness of synchronous programs or analysing cyclic circuits use (variants of) Binary Decision Diagrams (BDDs), most model checkers nowadays rely on SAT-solvers instead [7]. Here, the method of IC3/Property Directed Reachability (PDR) introduced by Bradley [5] is especially noteworthy. Therefore, part of the objective of this paper was to formulate causal correctness in such a way that advanced model checking procedures could be applied.

Chapter 2 gives a brief overview of existing methods for checking causality. Chapter 3 introduces the Quartz language and states our transformation of guarded actions to an equation system. In chapter 4 we present different options to create acyclic circuits from the equation system and assess their performance with respect to PDR in chapter 5. Chapter 6 closes the paper and outlines directions for future research.

2. Related Work

While early Esterel compilers V1 and V2 produce explicit automata, the first compiler based on logic networks (V3) does not allow cyclic dependencies at all [2]. Malik [11] uses ternary simulation to analyze combinational cyclic circuits. Here, the dual rail encoding of the cyclic circuit is represented using a BDD. By Kleene iteration on the BDD, an equivalent acyclic circuit is then computed. Shiple, Berry, and Touati [18] improve the iteration of the former by exploiting the nesting of cycles. To analyze sequential circuits, they first produce a BDD representation of the transition function and then utilize the model checking procedure from [8]. Their method has been used in the Esterel compiler V4 [2]. Although they use BDDs, their approach could be adapted to provide an unrolled circuit similar to what we do in section 4.1.

Schneider and Brandt [16] do not use an explicit transition function but translate guarded actions into a transition relation on the microstep level. They then formulate causal correctness as a property that has to hold infinitely often on all paths, i.e. G F in LTL. In [17], Schneider, Brandt, and Schuele describe a translation of Quartz or Esterel programs with delayed actions into an equation system with recursions on both the initial and the next output values.

3. Quartz

Quartz [15] is an imperative synchronous language developed by Klaus Schneider at the University of Kaiserslautern. As our method does not require any assumptions on the syntax of programs but will treat them on the level of guarded actions, we restrict ourselves to a brief overview of the Quartz language.

3.1. Overview

In general, Quartz is closely related to Esterel [3] and shares many of the underlying principles. Like in Esterel, execution is divided into macrosteps with the pause statement (among others). During a macrostep, all variables assume constant values that are determined through zero-time microsteps executed in parallel. When a variable is not explicitly assigned a value, its reaction to absence is triggered, i.e. depending on the storage class, it either receives a default value (`event`) or keeps its value from the last macrostep (`memorized`). In this regard, Quartz is more strict than Esterel as the latter allows variables to change values during one macrostep.

Variables can be declared as input (?x), output (!x) or inout (x). When a program is linked, inout becomes output. Nondeterminism is implemented through oracle variables, i.e. special input variables. Labels associated with the pause statement and other (redundant) statements like `halt` can be seen as event variables. The state of a program is thus given by the active labels, the values of memorized variables and the set of delayed assignments.

While Quartz allows variables of unbounded datatype, our analysis is currently limited to boolean programs, so all computations with bounded integers/naturals have to be formulated on bitvectors. For simplicity, we demand that all variables be scalar boolean.

The arguably most noticable deviation from Esterel–and the one with the most impact on the discussion here–is the introduction of assignments of variables instead of Esterel's `emit`. An `emit` in Esterel thus becomes a mere assignment of `true` (though `emit` exists for historical reasons). The possibility of write conflicts in Quartz programs thereby necessitates the usage of quaternary logic instead of ternary logic without $\top$.

Like in Esterel, assignments can be either immediate or delayed. Albeit they might give rise to a more compact compilation to software, delayed assignments behave just as immediate assignments in the next macrostep where the expression on the right-hand side is however evaluated in the current step. This is the central idea for the elimination of delayed assignments in section 3.4.

Both Quartz and Esterel allow the declaration of local variables. With a combination of preemption and loops containing a local declaration, it is possible that the scope of a local variable is left and entered during a single macrostep. Such programs are called schizophrenic and require more than one incarnation of the same variable. In the following we will be considering only fully linked programs (systems) as sets of guarded actions with all required variables (including reincarnated ones) known up front.

Example 3.1 (Zero-delay toggle) A toggle with zero delay could be implemented like in listing 1. Zero delay means that, when `t` is true, output `o` changes in the same macrostep, i.e. immediately. Table 3.1 shows a possible run. ♡

```
module toggle(bool ?t, event !o) {
    bool s;
    loop {
        if(t) {
            o = !s;
            next(s) = !s;
        }
        else {
            o = s;
        }
        l: pause;
    }
}
```

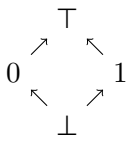
Listing 1: A toggle with zero delay

t	s	o
0	0	0
1	0	1
0	1	1
0	1	1
1	1	0

Table 3.1.: A possible run of the toggle

3.2. Quaternary Logic

Quaternary logic is the extension of conventional boolean logic to logic over the lattice $\overline{\mathbb{B}}$:



We extend the operators $\wedge$, $\vee$ and $\neg$ to $\overline{\mathbb{B}}$ by defining them as the (unique) monotonic functions satisfying

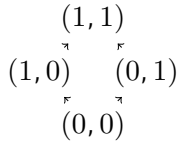
$$0 \wedge b = 0 = a \wedge 0, \quad 1 \wedge 1 = 1;$$
$$1 \vee b = 1 = a \vee 1, \quad 0 \vee 0 = 0;$$
$$\neg 0 = 1, \quad \neg 1 = 0$$

for all $a, b \in \overline{\mathbb{B}}$. By $\sqcup$ and $\sqcap$ we denote the join and meet operations over the lattice. The following conditional operator will be used extensively in the discussion of guarded actions:

$$a \mathbin{?} b := \begin{cases} b & 1 \leq a \\ \bot & \text{otherwise} \end{cases}$$

Note that all of the above operations are monotonic. This is essential for we now can define least fixpoints of functions over $\overline{\mathbb{B}}$ defined using those operations without worries.

We can express quaternary logic in convential boolean logic by *dual-rail encoding* observing that $\overline{\mathbb{B}}$ is isomorphic to $\mathbb{B} \times \mathbb{B}$:



The operators translate according to the following rules, effectively doubling the size of a formula:

$$\begin{aligned}
\neg(a_0, a_1) &= (a_1, a_0) \\
(a_0, a_1) \wedge (b_0, b_1) &= (a_0 \vee b_0, a_1 \wedge b_1) \\
(a_0, a_1) \vee (b_0, b_1) &= (a_0 \wedge b_0, a_1 \vee b_1) \\
(a_0, a_1) \mathbin{?} (b_0, b_1) &= (a_1 \wedge b_0, a_1 \wedge b_1) \\
(a_0, a_1) \sqcap (b_0, b_1) &= (a_0 \wedge b_0, a_1 \wedge b_1) \\
(a_0, a_1) \sqcup (b_0, b_1) &= (a_0 \vee b_0, a_1 \vee b_1)
\end{aligned}$$

3.3. Guarded Actions

Guarded actions serve as a convenient intermediate representation of Quartz programs and potentially programs written in other synchronous languages with least fixpoint semantics. Consider a system with input variables X, event variables Y_{event}, memorized variables $Y_{\text{memorized}}$ and labels Y_{label} and let $Y = Y_{\text{event}} \cup Y_{\text{memorized}} \cup Y_{\text{label}}$. All variables have default value $\text{Def}(y) = 0$ and inputs are always boolean, i.e. in $\{0, 1\}$. The behavior is then defined by a set of immediate guarded actions A_{imm} of the form $\phi \Rightarrow y = \alpha$ and a set of delayed guarded actions A_{del} of the form $\chi \Rightarrow \text{next}(y) = \beta$ where $y \in Y$ and $\phi, \chi, \alpha, \beta$ are boolean expressions over $X \cup Y$. Actions for $y \in Y_{\text{label}}$ may only take the form $\phi \Rightarrow \text{next}(y) = 1$, i.e. control flow is defined acyclic and therefore does not suffer from causality issues.

In the following, $M \in \mathbb{B}^{Y_{\text{memorized}}}$ denotes the values of memorized variables from the last macrostep, $I \in \mathbb{B}^X$ input values, $R \in \overline{\mathbb{B}}^Y$ the result of a reaction. Further, $\gamma_{I,M,R}$ denotes

the evaluation of expression γ under I, M, R over $\overline{\mathbb{B}}$. We define the absence value of variable $y \in Y$ as:

$$\mathrm{Abs}_M(y) = \begin{cases} M(y) & y \in Y_{\mathsf{memorized}} \\ \mathrm{Def}(y) & \text{otherwise} \end{cases}$$

The following function gives the result of a single round of applications of a set of immediate guarded actions A:

$$f^A_{I,M} : \overline{\mathbb{B}}^Y \mapsto \overline{\mathbb{B}}^Y, f^A_{I,M}(R)(y) = \left(\bigsqcup_{(\phi \Rightarrow y = \alpha) \in A} \phi_{I,M,R} \mathbin{?} \alpha_{I,M,R} \right) \sqcup \left(\left(\bigwedge_{(\phi \Rightarrow y = \alpha) \in A} \neg \phi_{I,M,R} \right) \mathbin{?} \mathrm{Abs}_M(y) \right) \tag{3.1}$$

The first part collects the the results of the assignments whose guards are known true while the second describes the reaction to absence, i.e. all guards are known false.

We are now ready to state the behavior of the program in pseudocode:

$M(y) \leftarrow 0$ `for` $y \in Y_{\mathsf{memorized}}$
$D \leftarrow \emptyset$ `// delayed assignments from previous step`
`loop:`
    `read inputs into` I
    `let` $A = A_{\mathsf{imm}} \cup D$
    `compute the least fixpoint` $R = \mu R. f^A_{I,M}(R) = \bigsqcup_i (f^A_{I,M})^i(\bot^Y)$
    $M(y) \leftarrow R(y)$ `for all` $y \in Y_{\mathsf{memorized}}$
    `assert` $R(y) \in \mathbb{B}$ `for` $y \in Y$ `// constructiveness`
    $D \leftarrow \{1 \Rightarrow y = \beta_{I,M,R} \mid (\chi \Rightarrow \mathrm{next}(y) = \beta) \in A_{\mathsf{del}} \wedge \chi_{I,M,R} = 1\}$

Algorithm 1: Execution of a Quartz program given as guarded actions

Example 3.2 (Zero-delay toggle cont'd) The toggle from example 3.1 could be translated to the following guarded actions with event variable r indicating that the program is running (remember that all variables are false by default):

$$\begin{aligned} \mathsf{true} &\Rightarrow \mathrm{next}(\mathsf{r}) = \mathsf{true} \\ \neg \mathsf{r} \vee \mathsf{l} &\Rightarrow \mathrm{next}(\mathsf{l}) = \mathsf{true} \\ \mathsf{t} \wedge (\neg \mathsf{r} \vee \mathsf{l}) &\Rightarrow \mathsf{o} = \neg \mathsf{s} \\ \mathsf{t} \wedge (\neg \mathsf{r} \vee \mathsf{l}) &\Rightarrow \mathrm{next}(\mathsf{s}) = \neg \mathsf{s} \\ \neg \mathsf{t} \wedge (\neg \mathsf{r} \vee \mathsf{l}) &\Rightarrow \mathsf{o} = \mathsf{s} \end{aligned}$$

♡

3.4. Equation System from Guarded Actions

The objective of this section is to transform a program given as a set of guarded actions as above into an equation system/sequential cyclic circuit of the form

$$\begin{cases} \vec{y} = f(\vec{x}, \vec{z}, \vec{y}) \\ \mathsf{init}(\vec{z}) = \vec{z}_0 \\ \mathsf{next}(\vec{z}) = g(\vec{x}, \vec{z}, \vec{y}) \end{cases} \tag{3.2}$$

with behavior

$$\begin{cases} \vec{y}^{(k)} = \mu\vec{y}.f(\vec{x}^{(k)}, \vec{z}^{(k)}, \vec{y}) \\ \vec{z}^{(0)} = \vec{z}_0 \\ \vec{z}^{(k+1)} = g(\vec{x}^{(k)}, \vec{z}^{(k)}, \vec{y}^{(k)}) \end{cases} \tag{3.3}$$

where $\vec{x}$ is the boolean input, $\vec{y}$ the (quaternary) boolean reaction, $\vec{z}$ the boolean state and f a monotonic function. In this context, it is natural to express causal correctness as a safety property $\mathrm{G} \bigwedge_i (y(i) \in \mathbb{B})$. In dual-rail encoding, $y(i) \in \mathbb{B}$ is equivalent to $y(i)_0 \oplus y(i)_1$.

For readability, we will not give a full formal description of the equation system but transform each variable with its guarded actions into one or more components of the state or reaction vectors. Clearly, input variables will be the same.

Eliminate delayed assignments When $y \in Y$ has both immediate and delayed actions, we can get rid of the delayed actions by introduction of two event variables t, v carrying whether delayed actions have triggered resp. their collective value: For each action $(\chi \Rightarrow \mathsf{next}(y) = \beta) \in A_{\mathsf{del}}$ create actions $\chi \Rightarrow \mathsf{next}(t) = 1$ and $\chi \Rightarrow \mathsf{next}(v) = \beta$. Then, replace all delayed actions for y with the immediate action $t \Rightarrow y = v$.

Immediate assignments only When y has no delayed actions, we add reaction and state variables as follows: For memorized y, add a state variable $\hat{y}$ that holds the absent value $\mathsf{Abs}(y)$: $\mathsf{init}(\hat{y}) = \mathsf{Def}(y)$ and $\mathsf{next}(\hat{y}) = y$; for event y, the absent value is the default value, i.e. 0. Similar to eq. (3.1), we then have for a reaction variable y:

$$y = \Big(\bigsqcup_{(\phi \Rightarrow y = \alpha) \in A_{\mathsf{imm}}} \phi \mathbin{?} \alpha \Big) \sqcup \Big(\big(\bigwedge_{(\phi \Rightarrow y = \alpha) \in A_{\mathsf{imm}}} \neg\phi \big) \mathbin{?} \mathsf{Abs}(y) \Big) \tag{3.4}$$

Delayed assignments only When y has only delayed actions, we add reaction and state variables as follows: the next state can be computed on a reaction variable y' similar to eq. (3.4):

$$y' = \Big(\bigsqcup_{(\chi \Rightarrow \mathsf{next}(y) = \beta) \in A_{\mathsf{del}}} \chi \mathbin{?} \beta \Big) \sqcup \Big(\big(\bigwedge_{(\chi \Rightarrow \mathsf{next}(y) = \beta) \in A_{\mathsf{del}}} \neg\chi \big) \mathbin{?} \mathsf{Abs}(y) \Big). \tag{3.5}$$

Regardless of whether y is memorized or event, we need a state variable y carrying the current value of y, i.e. $\mathsf{init}(y) = 0$ and $\mathsf{next}(y) = y'$. Further, y holds the absent value $\mathsf{Abs}(y)$ when y is memorized; otherwise, the absent value is $\mathsf{Def}(y)$.

Control flow When y is a label, i.e. all actions are delayed and of the form $\chi \Rightarrow \mathsf{next}(y) = 1$, eq. (3.5) could be simplified to

$$y' = \bigvee_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \chi$$

and $y' \in \mathbb{B}$ in the safety property for causal correctness of the program is redundant. However, we will not treat labels as a special case as it is not necessary in the chosen representation (see section 4.1).

4. Acyclic Circuit from Equation System

The last chapter described how causal correctness of a set of guarded actions can be expressed as a safety property G $\sigma(\vec{x}, \vec{z}, \vec{y})$ of a sequential cyclic circuit given by the equation system

$$\begin{cases} \vec{y} = f(\vec{x}, \vec{z}, \vec{y}) \\ \mathsf{init}(\vec{z}) = \vec{z}_0 \\ \mathsf{next}(\vec{z}) = g(\vec{x}, \vec{z}, \vec{y}) \end{cases} \tag{4.1}$$

with input $\vec{x} = (x_1, \ldots, x_{n_x}) \in \mathbb{B}^{n_x}$, reaction $\vec{y} = (y_1, \ldots, y_{n_y}) \in \mathbb{B}^{n_y}$ and state $\vec{z} = (z_1, \ldots, z_{n_z}) \in \mathbb{B}^{n_z}$ and f monotonic. Note that $\vec{y}$ was quaternary boolean in the last chapter; $\vec{y} \in \mathbb{B}^{n_y}$ can be easily achieved through dual-rail encoding. The semantics are still:

$$\begin{cases} \vec{y}^{(k)} = \mu\vec{y}.f(\vec{x}^{(k)}, \vec{z}^{(k)}, \vec{y}) \\ \vec{z}^{(0)} = \vec{z}_0 \\ \vec{z}^{(k+1)} = g(\vec{x}^{(k)}, \vec{z}^{(k)}, \vec{y}^{(k)}) \end{cases} \tag{4.2}$$

As to our knowledge no safety checker supports circuits of above form, we have to derive an equivalent safety property on an acyclic sequential circuit, i.e. with no $\vec{y}$, in order to use existing checkers.

4.1. Unrolling

With some $f'(\vec{x}, \vec{z}) = \vec{y}$ we would have an equivalent acyclic circuit and could leave the safety property unchanged. One option is Kleene iteration:

$$\vec{y} = \mu\vec{y}.f(\vec{x}, \vec{z}, \vec{y}) = \bigsqcup_i f(\vec{x}, \vec{z}, \bot^{n_y}) = \bigsqcup_{i \leq k} f(\vec{x}, \vec{z}, \bot^{n_y})$$

for some $k \leq n_y$ as $\vec{y}$ has n_y components and each component can increase at most once. With no further inspection of the function, the best bound for k would probably be the length of the longest dependency chain on the y_i. Unfortunately, computing the length of the longest dependency chain is NP-hard by reduction from the directed Hamiltonian path problem.

Kleene iteration is easily implemented when f is stored in (binary) decision diagrams–in fact, Malik [11] takes this route. However, BDDs can grow exponentionally large on reasonable inputs, e.g. on a single-macrostep multiplier, where it is not necessary to have a unique representation (the multiplier is acyclic thus causally correct). Circuits would naturally be more robust

a representation; nonetheless, naive Kleene iteration produces large circuits ($n \cdot |f|$) with (usually) large portions not being needed for the final output: most practical programs probably do not contain large instantaneous cycles so that the fixpoint would be reached in far fewer than n iterations. Furthermore, not every variable would require the same number of iterations; some variables might not depend on themselves so a fixed ordering of those would be sufficient. Shiple, Berry, and Touati [18] improve the Kleene iteration by first iterating smaller subcycles. Albeit their algorithm uses BDDs, adapting it for circuits might still yield a useful method. Before we can continue with the presentation of our approach, we should introduce the exact notion of circuit that we will work with:

And-Inverter Graphs (AIGs) are directed acyclic graphs/circuits consisting of input nodes (including constant false) and (two-input) AND-gates with inverters on the edges. They will serve us as a convenient compact representation of boolean functions and play an integral part in our algorithm by merging duplicate parts of the output. Although more powerful reductions can be applied during construction such as merging functionally equivalent nodes (functionally reduced AIGs (FRAIGs) [12]), single-gate structural hashing is straightforward to implement and sufficient for the approaches described in this chapter: before adding a new gate it is first checked whether a gate with the same input edges (up to permutation) has already been constructed. This can efficiently be accomplished by means of a hashtable. Further, the following reductions are applied before any node is added (constant propagation):

$$0 \wedge a = a \wedge 0 = a \wedge \neg a = \neg a \wedge a = 0$$
$$\neg 0 \wedge a = a \wedge \neg 0 = a \wedge a = a$$

Structural hashing provides several benefits while nodes can still be added in near-constant time: Although de-duplication happens on a lower level, it gives rise to a sorting of the guarded actions in the algorithm below (when the circuit has been derived from guarded actions). That is, it is no disadvantage that the following algorithm does not work on the level of guarded actions but on the level of variables: actions that are known early in the evaluation will get merged through structural hashing even though other action might need to be duplicated. Furthermore, labels are detected automatically and dual-rail encoding of their equations is very cheap: consider label y with equations

$$\begin{aligned} \mathsf{init}(y) &= 0 \\ \mathsf{next}(y) &= y' \\ y' &= \Big(\bigsqcup\nolimits_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \chi \; ? \; 1 \Big) \sqcup \Big(\big(\bigwedge\nolimits_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \neg \chi \big) \; ? \; 0 \Big). \end{aligned}$$

In dual-rail encoding we have after constant propagation:

$$\begin{aligned} y'_1 &= \bigvee\nolimits_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \chi_1 \\ y'_0 &= \big(\bigwedge\nolimits_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \neg \chi \big)_0 = \neg \bigvee\nolimits_{(\chi \Rightarrow \mathsf{next}(y)=1) \in A_{\mathsf{del}}} \chi_1 \end{aligned}$$

so $y'_0 \oplus y'_1 = 1$ is not added to the safety property and only χ_1 has to be computed.

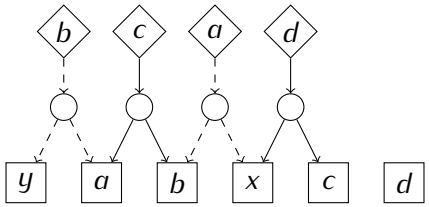


Figure 4.1.: AIG from example 4.1 before unrolling

Example 4.1 (AIG from equation system) Consider the following combinational equation system with inputs $x, y \in \mathbb{B}$ and reaction variables $a, b, c, d \in \mathbb{B}$.

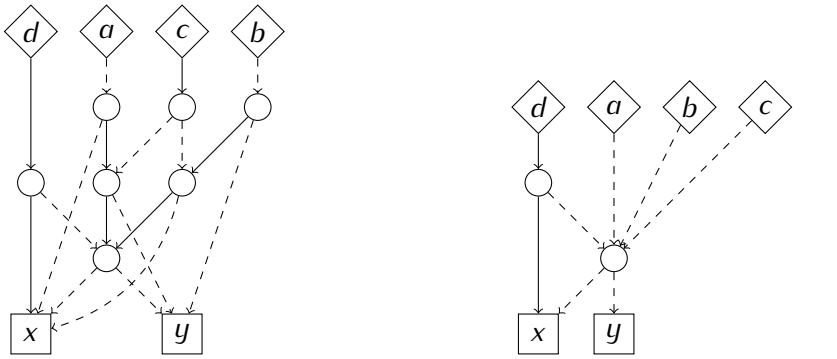
$$\begin{aligned} a &= x \vee b \\ b &= y \vee a \\ c &= a \wedge b \\ d &= x \wedge c \end{aligned}$$

Figure 4.1 shows the AIG representation of the equations. In the AIG, reaction variables become inputs (rectangles) while the respective right-hand sides become outputs (diamonds); inverted edges are shown dashed. ♡

Top-Down Unrolling The fundamental difference of our method compared to [11, 18] is that ours computes the unrolling from the end while they produce the unrolling in a forward manner. In this, it is very similar to the local method of model checking μ-calculus [19], which also was the main source of inspiration. Basically, we compute an acyclic circuit for each y_i as follows: First, the AIG of f_i is traversed. When we reach another y_j, we plug in f_j and continue recursively; if however y_i is encountered, we plug in false instead.

While this approach does not work for BDDs, it is easily implemented when f is presented as an AIG: then, $\vec{x}$, $\vec{z}$ and $\vec{y}$ would be nodes and the components $f_1, \ldots, f_n$ of f corresponding to $y_1, \ldots, y_n$ edges. We would now replace all y_i in the equation system with $y_i^\emptyset$; here, u^A stands for the node/edge u where all y_j with $j \in A$ have already been encountered. The $y_i^\emptyset$ are recursively given as:

$$\begin{aligned} y_i^A &= \begin{cases} \text{false} & \text{if } i \in A \\ f_i^{A \cup i} & \text{otherwise} \end{cases} \\ x_i^A &= x_i \\ z_i^A &= z_i \\ (\neg u)^A &= \neg u^A \\ (u \wedge v)^A &= u^A \wedge v^A \end{aligned}$$



(a) Using Kleene iteration (b) Using the top-down approach

Figure 4.2.: Unrolled AIGs from example 4.1

All of the above results on the right-hand side are of course created using the AIG. As the original input is stored in an AIG as well, it seems natural to use a hierachy of caches (one for each A; created on demand) for already completed unrollings. Further, it might be advantageous to first query the caches for subsets (this does not change the fixpoint due to monotonicity) and insert the result in cache for the smallest set of y_i that actually were replaced with false. To do this efficiently, some ordering of nodes will probably be necessary so not all caches corresponding to all the subsets of a given A will be queried but only linearly many.

Example 4.2 (Unrolled AIG from equation system) Unrolling the equation system from example 4.1 using the top-down approach yields a much smaller circuit (fig. 4.2b) than Kleene iteration (fig. 4.2a): Kleene iteration continues to duplicate a and b when they are already stable while the top-down unrolling does not plug in the same right-hand side twice on the same path. To this end, the top-down approach explores the paths:

$$ab \quad ba \quad cab \quad cba \quad dcab \quad dcba$$

Hence, it requires 13 different sets A. ♡

The correctness of the top-down approach follows immediately from Bekič' lemma [1] (for the short proof of the stated variant, see e.g. [20]) and the fact that fixpoints on single boolean variables are reached in one iteration.

Lemma 4.3 (Bekič) *Let $f : D \times E \to D$, $g : D \times E \to E$ be monotone functions and D, E complete lattices. Then*

$$\mu(x, y).(f(x, y), g(x, y)) = (\mu x.f(x, \mu y.g(x, y)), \mu y.g(\mu x.f(x, y), y)). \qquad \square$$

To see this, consider for instance y_1 (so notation stays manageable):

$$y_1 = (\mu \vec{y}.f(\vec{x}, \vec{z}, \vec{y}))_1$$

By Bekič' lemma:

$$= \mu y_1.f_1(\vec{x}, \vec{z}, (y_1, \mu y_2, \ldots, y_n.f_{2,\ldots,n}(\vec{x}, \vec{z}, \vec{y})))$$

And as the fixpoint is reached in one step:

$$= f_1(\vec{x}, \vec{z}, (\bot, \mu y_2, \ldots, y_n.f_{2,\ldots,n}(\vec{x}, \vec{z}, (\bot, y_2, \ldots, y_n))))$$

The top-down approach computes a minimal-delay unrolling in the sense that no y_i is computed twice on any path. However, this comes at the price of the runtime being exponential in n in the worst case (e.g. when every variable directly depends on every other). Nevertheless, this seems adequate as the algorithm effectively solves the problem of finding the longest dependency chain on the y_i. Furthermore, it seems fairly unlikely to see the worst case at scale as gates will always have non-zero delay in actual implementations so arbitrarily long cycles do not make practical sense. Regarding the size of the output, hope lies mainly on the AIG.

4.2. Fixpoint in Steps

A more space efficient alternative in the presence of many cyclic dependencies would be to simulate Kleene iteration in the circuit. The safety property $\mathrm{G}\,\sigma$ would then be changed to $\mathrm{G}(\text{fixpoint reached} \wedge \sigma)$. As the input must not change during the simulation, latches for holding the inputs are needed together with some initialization logic.

Concretely, there is a latch for every input, state and reaction variable (all latches are initialized to 0). Further, there is a latch for indicating startup, i.e. has value 1 in every but the very first step. In the first step and whenever the fixpoint has been reached (the next value implies the current value for each reaction variable), inputs are read into their latches, the state variables are updated and the latches for the reaction variables are reset. Otherwise, all inputs and state variables are held constant and the reaction variables are updated with the value of the right-hand-side of their respective equation. Of course, the safety property is not asserted in the first step.

Clearly, this approach requires linearly many operations on the AIG and the resulting acyclic circuit is larger by only a constant factor.

5. Evaluation

5.1. Implementation

The Averest framework[1] for the Quartz language is used to compile a set of Quartz modules into a system of guarded actions. Those are then read in using Averest and converted into an equation system as described in section 3.4. The resulting sequential cyclic circuit is then converted into a sequential acyclic circuit. The acyclic circuits are then output in binary AIGER format [4]. The safety property (after the fixpoint has been reached, all variables shall have boolean values) is encoded in the single outputs of the acyclicized circuits.

AIGER has been chosen as the output format because it is the official input format of the Hardware Model Checking Competition (HWMCC), meaning that all competing model checkers (most of which are available freely) could potentially be used to check the models produced. Because Averest is written in F#, we chose to implement our translation to AIGER, starting with the guarded actions, entirely in F#, too. As AIGs are an integral part of the proposed method but there was no implementation available for F#, we had to write a minimalistic AIG manager ourselves, providing only basic structural hashing, i.e. constant propagation and merging identical gates, and a method for writing (binary) AIGER. Given a unified way to handle large propositional formulas memory-efficiently, it seemed natural to use a single AIG for representing all formulas starting with those read from the guarded actions.

5.2. Comparison

In all below test cases we compare the following methods for making the circuit acyclic:

stepped Simulation of Kleene iteration in steps as described in section 4.2.

unrolled Kleene Naive Kleene iteration with fixed n iterations.

unrolled Bekič Top-down unrolling with one cache per set A (created on demand) and no further improvements (see section 4.1).

Additionally, each of the resulting circuits was converted into a FRAIG and verified independently (denoted by "+ fraig").

[1] `averest.org`

We chose the toolkit ABC [6] due to its overall good verification performance in the HWMCC 2014 [7], it having a well documented implementation of PDR [9] and it using AIGs as primary representation of models during verification. Further, ABC supports the creation of FRAIGs from existing AIGs or on-the-fly. All tests have been carried out on a machine with an Intel® i7-3720QM and 8 GB of RAM running Linux with Mono as the .NET implementation. We used the latest version of ABC as of 2019-03-08. Both the `fraig` and `pdr` commands of ABC were run with default options.

5.3. Test Cases

5.3.1. Rivest

Rivest [14] gave an example of a cyclic combinational circuit for which any acyclic circuit computing the same functions will be larger (in terms of two-input gates) by a factor of at least $\frac{3}{2}$.

```
module Rivest(event [N]bool ?x, event [2*N]bool y) {
    loop{
        y[0] = x[0] & y[2*N - 1];
        y[1] = x[1%N] | y[0];
        for(i = 0..N-2) {
            y[2*i + 2] = x[(2*i + 2)%N] & y[2*i + 1];
            y[2*i + 3] = x[(2*i + 3)%N] | y[2*i + 2];
        }
        pause;
    }
}
```

Listing 2: Quartz implementation of Rivest's circuit for variable N

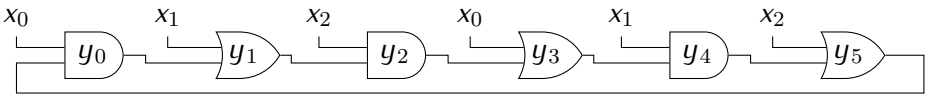
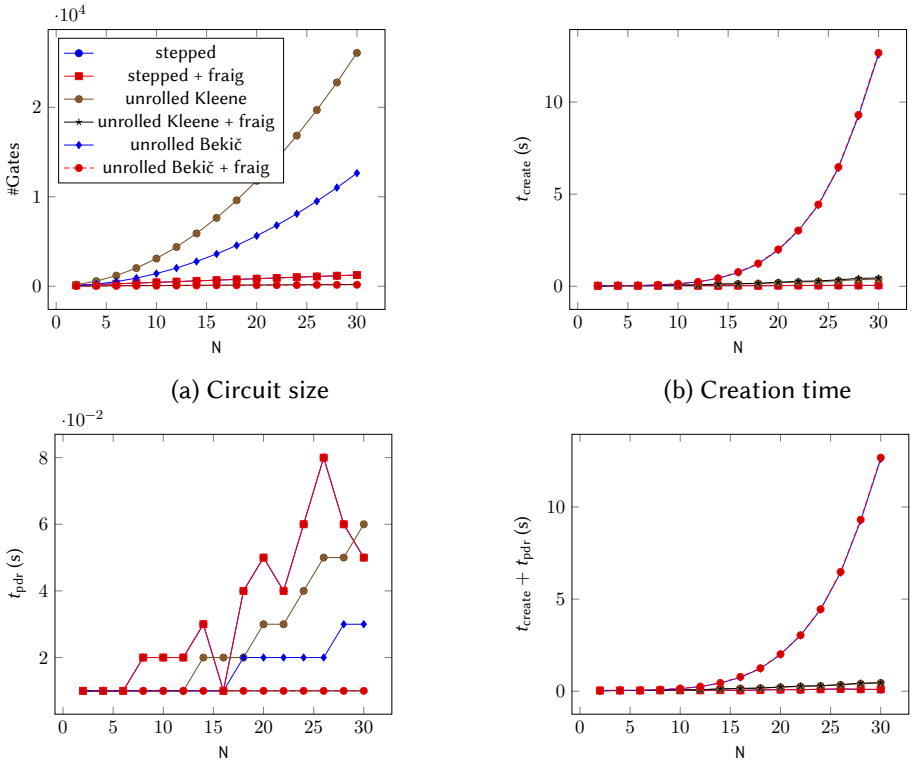


Figure 5.1.: Rivest's circuit for $N = 3$

First of all, notice that the circuits derived from the Quartz code are combinational, i.e. reachability could be ignored. (Actually, the program has two states (with equivalent output behavior however) due to two labels produced.) Further, notice that the circuit is causally correct only for odd N. To aid readability, we have thus split the results for odd and even N (see figs. 5.2 and 5.3). In both cases however, the picture is mostly the same:

Naturally, the circuit produced by Kleene iteration grows rather quickly (quadratic) while the circuit produced by the top-down approach stays at about half the size of the former. However, the top-down approach becomes increasingly slow for larger N (still not exponential) where naive Kleene iteration still has an acceptable runtime. The circuit simulating Kleene iteration in steps on the other hand has neglectable size and is produced just as quickly (linear in N).

Without functional reduction, the runtime of PDR on the unrolled circuits seems to be related to the circuit size but grows quicker. Generally, counterexamples are found much quicker (by a factor of 25) than circuits are verified. As one would expect from the fact that the circuits are mostly combinational, the unrolled circuits are drastically smaller after functional reduction (the output even becomes independent from the inputs in the causally correct case) and are then checked almost instantly. The stepped circuit by contrast does not benefit from functional reduction at all–it remains unchanged. While the runtime of pdr on the unrolled circuits grows consistently, the stepped circuit shows great variation in that regard and generally takes the most time to verify among the approaches. For even larger N (up to 63) however, the stepped circuit is checked in about the same time as the unrolled circuit from the top-down approach while the unrolled circuit by Kleene iteration takes twice as long. Nonetheless, the creation time of the top-down unrolling quickly becomes unacceptable.



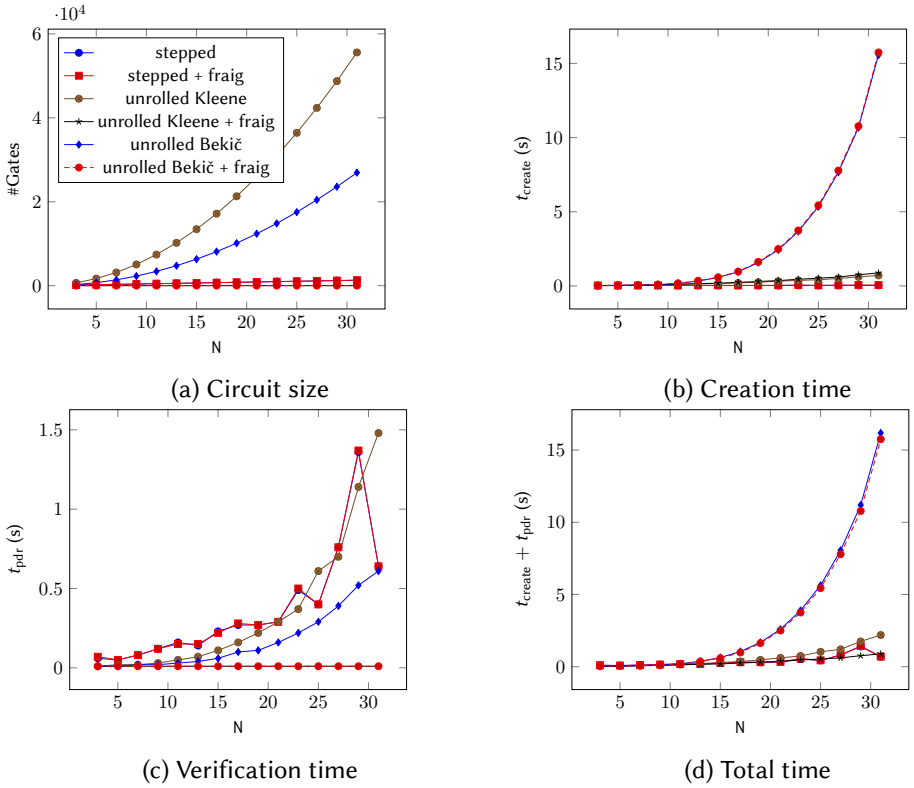
(a) Circuit size

(b) Creation time

(c) Verification time

(d) Total time

Figure 5.2.: Results for causally incorrect Rivest circuits (even N)



(a) Circuit size

(b) Creation time

(c) Verification time

(d) Total time

Figure 5.3.: Results for causally correct Rivest circuits (odd N)

5.3.2. Arbiter

Listing 3 shows the implementation of a token-ring arbiter for N processes that works as follows: Processes may request access to the bus in the next macrostep; the process that holds the token at the beginning of the macrostep is allowed access to the bus but has to pass on the token then, i.e. has lowest priority for being allowed access in the next step. When no processes request access, the token will not move.

While Rivest's circuits were combinational, reachability is now essential: at any instant, at most one process must have the token. Clearly, there are N reachable states (not counting the initial state which is redundant). Nonetheless, the results are comparable (see fig. 5.4):

Not surprisingly, Kleene iteration produces even larger circuits due to the increased complexity of the example; the top-down unrolling on the other hand stays manageable (in terms of size and creation time) for the shown N—while its runtime still deteriorates quickly, it is even lower than for Rivest's circuits of the same N despite the increased complexity of the program. Again, the stepped circuits are already functionally reduced whereas fraiging the unrollings yields substantial reductions in size. Interestingly, the funcionally reduced Kleene unrolling seems to grow only linearly while the unrolling using Bekič' lemma grows faster than linear.

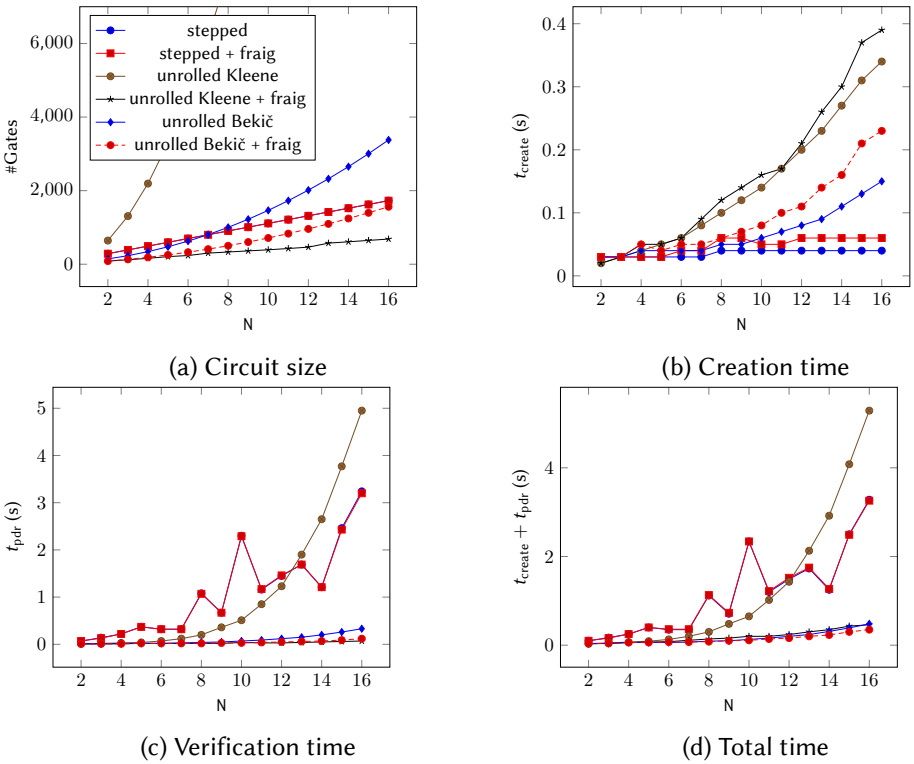
```
module arbiter(
    [N]bool ?req, [N]bool ?data,
    bool !valid, bool !out
) {
    event [N]bool pass;
    event [N]bool token;

    token[0] = true;

    loop {
        for(i = 0..N-1) {
            pass[i] =
                (pass[(i + N - 1)%N] & !req[i]) | token[i];
            if(token[i]) {
                valid = req[i];
                out = data[i];
            }
            next(token[i]) =
                (token[i] | req[i]) & pass[(i + N - 1)%N];
        }
        l: pause;
    }
}
```

Listing 3: Quartz implementation of the Arbiter

As for the time taken by PDR, the stepped circuit once more shows great variation in this regard. For the unrollings, verification time still seems to be related to circuit size and functional reduction is able to lower verification time greatly. Despite the quickly deteriorating runtime of the top-down unrolling, it still performs rather well for the considered range of N.

5.3.3. AsyncArbiterBoch82

This test case is an implementation of an asynchronous arbiter that was taken from the Averest distribution. The results are shown in table 5.1. The size of the stepped circuit is probably attributed to the large number of guarded actions; their complexity seems to be reduced in the unrollings, resulting in smaller circuits. Once again, fraiging leaves the stepped circuit unchanged while the unrollings are reduced to less than the half.



(a) Circuit size

(b) Creation time

(c) Verification time

(d) Total time

Figure 5.4.: Results for Arbiter

Method	#Gates	t_{create} (s)	t_{pdr} (s)	$t_{\text{create}} + t_{\text{pdr}}$ (s)
stepped	2066	0.05	91.28	92.98
stepped + fraig	2066	0.12	92.93	93.05
unrolled Kleene	641	0.03	0.19	0.22
unrolled Kleene + fraig	276	0.04	0.08	0.12
unrolled Bekič	641	1.10	0.19	1.29
unrolled Bekič + fraig	276	1.11	0.08	1.19

Table 5.1.: Results for AsyncArbiterBoch82

6. Conclusion & Outlook

In this paper we have seen how a synchronous program given as a set of guarded actions (both immediate and delayed) can be translated into an equation system describing a synchronous circuit with cyclic combinational part. Our main contribution here was the usage of the lattice-join operation to merge the effects of several guarded actions and the introduction of a special conditional operator to describe the result of a single immediate assignment. We then demonstrated several options for transforming the cyclic circuit into an acyclic one that could then be checked by a large variety of existing model checkers. To that end, we developed a top-down unrolling, drawing inspiration from the local method of model checking μ-calculus. Although each of the above steps probably needs to be refined to provide an overall satisfactory algorithm for checking constructiveness of very large programs, the combination of compiling a synchronous program into an acyclic circuit and SAT-based safety checking could potentially be scalable enough for this task.

Unfortunately, no representative large scale examples were available to compare the different methods for making the circuit acyclic. However, the examples we could test all were somewhat pathological: they all basically consist of a single large cycle that results in large unrollings. Therefore, we could still draw some conclusions:

First of all, considering microsteps does not help in the reachability analysis: despite being fast to generate and having guaranteed linear size, verification of the circuit that simulated the fixpoint was generally slow. We further noted that in all tested examples, the resulting circuit was already functionally reduced–not only do microsteps increase the state space, they result in a complicated transition function as well. Thus, in some cases, the stepped circuit was even larger than the unrolled ones. The added microsteps thus appear to be more a hindrance than an advantage when PDR is applied. In general, having a small state space seems well worth the effort.

What can be said about unrolling the fixpoint then? Here, the size of the circuit seems to be the determining factor. The top-down performed rather well in that regard–if only its worst-case runtime could be improved. Albeit being much faster in the worst case, naive Kleene iteration produces prohibitively large circuits in general. By construction, the top-down approach will work best when the variables have tree-like dependencies and it will get slower as the dependency graph gets denser. In that case, computing the fixpoint in a bottom-up manner like Kleene iteration will be faster and should result in similar circuits, too.

Although we did not intend to use functional reduction for the unrolling at first, we still applied it to the already unrolled circuits and were surprised of its speed and yield in terms of size and thereby reduced verification time. While both Kleene iteration and the top-down approach

benefitted substantially, in one instance it actually was Kleene iteration that resulted in smaller reduced circuits. It that case, the minimum-delay property of the top-down unrolling might have lead to circuits that are too 'diverse'. Thus, a combination of Kleene iteration and a SAT-based detection of when a variable has stabilized might be a good starting point for a more efficient circuit-based unrolling. One could then even consider improvements of the iteration like in [18] that would require BDDs otherwise.

So far, we have only considered boolean programs. While this assumption is reasonable when the program is intended to be compiled to hardware (so one has to allocate arithmetic units anyway), bitblasting a program that is to be compiled to software (so arithmetic operations can be used freely) will likely result in circuits that are too large to be verified directly with methods designed for boolean programs. Still, under the assumption that causality is mostly a property of control flow and the dependencies among variables, abstract interpretation might be able to reduce the problem to a problem on booleans: assigning a complex variable twice in the same macrostep is likely to produce a write conflict; thus, it should be sufficient to detect mutually exclusive choices and make assignments of non-boolean variables nondeterministic. Abstract interpretation could now either be done before compiling the program to a circuit or one could compile the program to software (on the level of guarded actions) and abstract then (or use a software model checker). This way, all the safety checking methods for finite state systems still apply. Methods for infinite state transition systems would however become necessary for programs with dynamic array accesses.

Considering how close synchronous programs and logic programs are related, exploring in this direction might also be worthwhile. Especially the computation of fixpoints seems to be treated extensively in the literature of the latter.

Finally, even though causality analysis seems to be mechanizable to quite some extent, more work is required on the language front to make the analysis modular. The definition of causality interfaces could be one solution for modules with rather small interfaces; for modules with complex I/O behavior however, there might be no way around a global causality analysis.

Bibliography

[1] Hans Bekič. "Definable operations in general algebras, and the theory of automata and flowcharts". In: *Programming Languages and Their Definition*. Springer, 1984, pp. 30–55.

[2] Albert Benveniste et al. "The Synchronous Languages 12 Years Later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[3] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In: *Science of computer programming* 19.2 (1992), pp. 87–152.

[4] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Oct. 2007.

[5] Aaron R Bradley. "Understanding ic3". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 1–14.

[6] Robert Brayton and Alan Mishchenko. "ABC: An academic industrial-strength verification tool". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40.

[7] Gianpiero Cabodi et al. "Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2016), pp. 135–172.

[8] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. "Verification of Synchronous Sequential Machines Based on Symbolic Execution". In: *International Conference on Computer Aided Verification*. Springer. 1989, pp. 365–373.

[9] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2011, pp. 125–134.

[10] Nicolas Halbwachs. "A synchronous language at work: the story of Lustre". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE. 2005, pp. 3–11.

[11] Sharad Malik. "Analysis of Cyclic Combinational Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.7 (1994), pp. 950–956.

[12] Alan Mishchenko et al. *FRAIGs: A unifying representation for logic synthesis and verification*. Tech. rep. ERL Technical Report, 2005.

[13] Marcus D. Riedel. "Cyclic Combinational Circuits". PhD thesis. California Institute of Technology, 2004.

[14] Ronald L. Rivest. “The Necessity of Feedback in Minimal Monotone Combinational Circuits”. In: *IEEE Transactions on Computers* 6 (1977), pp. 606–607.

[15] Klaus Schneider. *The synchronous programming language Quartz*. Tech. rep. Department of Computer Science, University of Kaiserslautern, 2009.

[16] Klaus Schneider and Jens Brandt. “Performing Causality Analysis by Bounded Model Checking”. In: *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*. IEEE. 2008, pp. 78–87.

[17] Klaus Schneider, Jens Brandt, and Tobias Schuele. “Causality Analysis of Synchronous Programs with Delayed Actions”. In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2004, pp. 179–189.

[18] Thomas R Shiple, Gerard Berry, and Herve Touati. “Constructive Analysis of Cyclic Circuits”. In: *Proceedings ED&TC European Design and Test Conference*. IEEE. 1996, pp. 328–333.

[19] Colin Stirling and David Walker. “Local model checking in the modal mu-calculus”. In: *Theoretical Computer Science* 89.1 (1991), pp. 161–177.

[20] Glynn Winskel. “Techniques for recursion”. In: *The Formal Semantics of Programming Languages: An Introduction*. MIT press, 1993. Chap. 10.

A. Time Complexity

Not taking reachability into account, checking causality of boolean programs, i.e. checking whether all outputs are boolean for all states, is equivalent to checking the validity of a propositional boolean formula and thus co-NP-complete.

Taking reachable states into account, one easily sees that

Theorem A.1 *Checking causal correctness of boolean programs is PSPACE-hard.* ♢

Proof. Universality of an FSM is reduced to checking causality: Given a possibly nondeterministic FSM $A = (Q, \Sigma, \Delta, x_0, F)$ where, wlog., $Q = \{x_0, \ldots, x_n\}$ and $\Sigma = \mathcal{P}(\{a_0, \ldots, a_m\})$, the question is whether $\mathcal{L}(A) = \Sigma^*$. The following Quartz program performs an on-the-fly subset-construction. Universality of A is thus equivalent to the property: in every step, a final state is in the current set of states. The program is causally correct iff this property holds on every input sequence.

```
module universal(event ?a_0, ..., ?a_m, x_0, ..., x_n, err) {
  x_0 = true;
  loop {
    if(¬ ⋁_{x∈F} x) err = !err;
    // for each x_k ∈ Q:
    if(⋁_{(x_i,A,x_k)∈Δ} x_i ∧ A) next(x_k) = true;
    pause;
  }
}
```

□

and the proposed method gives:

Theorem A.2 *Checking causal correctness of boolean programs is in PSPACE.* □