Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 88 (2004) 55–69
www.elsevier.com/locate/entcs

Safety Property Verification of Cyclic Synchronous Circuits

Koen Claessen [1]

Department of Computing Science
Chalmers University of Technology
Gothenburg, Sweden

Abstract

Today's most common formal verification tools for hardware are unable to deal with circuits containing combinational loops. However, in the areas of hardware compilation, circuit synthesis and circuit optimization, it is quite natural for a subclass of these loops, the so-called *constructive* loops, to arise. These are loops that physically exist in a circuit, but are never logically taken. In this paper, we present a method for safety property verification of circuits containing constructive combinational loops, based on propositional theorem proving and temporal induction. It can be used to just prove constructivess of circuits, but also to directly prove safety properties of the circuits. Unlike previously proposed methods, no fixed point iteration is needed, we do not have to compute reachable states, and no cycle-free representation of the circuit has to be computed.

1 Introduction

Synchronous circuits containing combinational loops often arise in in the areas of hardware compilation, circuit synthesis and circuit optimization. An example is the synchronous language Esterel, which can directly be compiled to hardware circuits that possibly contain cyclic logic [10]. Implementing the same functionality without the cyclic logic often means a blow-up in circuit size. Today's most common circuit analysis and verification tools however reject the use of combinational cycles in synchronous hardware. This makes

[1] Email: koen@cs.chalmers.se

1571-0661
doi:10.1016/j.entcs.2003.05.004

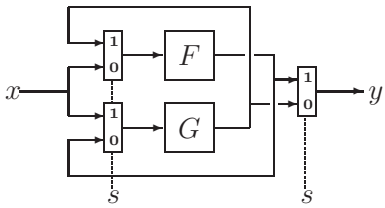

Fig. 1. A constructive cyclic circuit

it difficult to use standard tools in order to formally verify properties of the generated cyclic circuits.

However, often the produced cycles are so-called *false* cycles, in the sense that they do not really cause a problem when implementing and running the circuit electrically. An example taken from [7] (presented in Figure 1) consists of a cyclic circuit with two combinational subcircuits F and G. The circuit either computes $y = G(F(x))$ or $y = F(G(x))$, depending on the selection signal s. The circuit uses only one copy of F and G each and therefore contains a combinational cycle. However, the cycle is false in the sense that the output y is always well-defined.

For a given cyclic circuit, a separate analysis of the circuit is needed to prove that all present cycles are false, or, in the terminology of [10], that the circuit is *constructive*. In 1994, Malik presented such an analysis for combinational circuits [7], which was extended by Shiple et al. to be able to deal with sequential circuits [10]. Their analysis produces a new circuit, which is functionally equivalent to the original circuit, but does not contain any cycles. This new circuit can then be analyzed by other formal verification tools in order to check formal properties of the circuit.

The mentioned analyses are both based on BDDs [3]. This has as a drawback that circuits which are difficult to represent using BDDs are difficult to handle using the method, since the method inherently relies on computing a BDD which represents the function that the circuit implements. Also, the analyses involve a fixed point computation over BDDs, which might introduce extra costs.

In 1999, Namjoshi et al. presented a circuit analysis method that does not require a fixed point iteration [8]. However, their method only works for combinational circuits, and to make the method usable for sequential circuits they resort to similar methods as the ones in [10]; decision diagrams are used

to calculate the reachable states in the system.

The main contribution of this paper is the following. We present a sound and complete automatic analysis method for circuits containing combinational loops that does not involve a fixed point iteration, calculating reachable states, or computing a new cycle-free circuit, that proves constructiveness of circuits. We use the basic idea from [8], but we use a propositional theorem prover (sometimes called SAT-solver) instead of decision diagrams. Then, we use temporal induction [9] to extend the method to sequential circuits.

As said, the method does not compute a non-cyclic equivalent circuit. Instead, one can use the method directly to also prove possible safety properties of the circuit.

The rest of the paper is organized as follows. The first three sections can be seen as a tutorial on the subject and introduce background material, large parts of which are also presented elsewhere; in Section 2, we introduce the definition of circuits, and what their naive, classical semantics is; in Section 3, we show how to verify safety properties of these circuits, under the classical semantics, using a propositional theorem prover; in Section 4, we present the constructive semantics of circuits, which corresponds more closely to what happens in electrical circuits. Our main result is presented in Section 5, where we show how safety properties of cyclic circuits can be proved, under the constructive semantics. Finally, in Section 6 we discuss related work and conclude.

2 Classical Semantics of Circuits

In this section, we introduce the model of combinational and sequential circuits we use in the paper, and what their classical semantics is. We closely follow the terminology from [1].

Combinational Circuits. A *boolean formula* is built up from variables, x, y, z, and operators $\mathbf{0}, \mathbf{1}$ (nullary), $\neg$ (unary), and $\wedge, \vee, \Rightarrow, \oplus$ (binary). A *definition* is written $x = f$, and consists of a boolean variable x and a formula f.

A *combinational circuit* is a finite set C of definitions, such that, for every variable x, there is at most one definition of the form $x = f$ in C. The restriction on multiple definitions is added because we do not want to talk about circuits which contain points which are driven by multiple signals.

A variable x of a circuit C is called an *input*, if there is no definition of

the form $x = f$ contained in C. A *solution* of a combinational circuit is a valuation, i.e. an assignment of all variables to a boolean value **0** or **1**, such that all definitions are satisfied, using the usual interpretation of the boolean operators.

Sequential Circuits. A *delayed definition* is written $x := y$, and consists of two variables x and y. A *sequential circuit* (C, D) is a pair of a combinational circuit C and a finite set of delayed definitions D. Again, we apply the restriction that, for every variable x, there can be at most one definition of the form $x = f$ in C or a delayed definition of the form $x := y$ in D. A variable x is called an *input* of (C, D), if there is no definition in C of the form $x = f$ and no delayed definition in D of the form $x := y$.

In order to help defining the semantics of sequential circuits, we define the following renaming operations. Let $i \geq 1$ be a natural number. For a combinational circuit C, we write $C[i]$ for the copy of C, with each variable x replaced by a fresh variable x_i. For a set of delayed definitions D, we write $D[i]$ for the combinational circuit $\{x_i = y_{i-1} \mid x := y \in D\}$. The initial state of the circuit is dealt with by defining D_0 as the combinational circuit $\{y_0 = \mathbf{0} \mid x := y \in D\}$. Lastly, for a valuation s over variables x, we write $s[i]$ for the valuation over labelled variables x_i, for which it holds that $s[i](x_i) = s(x)$.

For a given sequential circuit (C, D), we define the *combinational expansion*, written $(C, D)[k]$, as the combinational circuit:

$$\bigcup_{i=1}^{k} C[i] \cup D[i]$$

A sequence of valuations $(s_1, \ldots, s_k)$ is called a *solution path* of a sequential circuit (C, D), if $(s_1[1] \cup s_2[2] \cup \cdots \cup s_k[k])$ is a solution of the combinational circuit $D_0 \cup (C, D)[k]$.

3 Proving Safety Properties

In this section, we show how to automatically prove safety properties of combinational and sequential circuits.

Safety Properties. For the purposes of this paper, a safety property is a particular variable p in the circuit that is supposed to be always true for all possible scenarios of a circuit. We say that, for a given combinational circuit

C, a safety property p is *valid*, if $s(p) = \mathbf{1}$, for all solutions s of C. If we want the safety property to be a more complicated property than just a variable, namely a formula f, we can simply add $p = f$ as a definition to C. In the rest of the paper, we will sometimes do this implicitly.

For a sequential circuit (C, D), we say that a safety property p is *valid*, if, for all k, for all solution paths $(s_1, \ldots, s_k)$ of (C, D), and for all $1 \leq i \leq k$, it is the case that $s_i(p) = \mathbf{1}$.

Proving Combinational Properties. In order to check if a given property p is valid for a given combinational circuit C, we can use a propositional logic theorem prover, often called SAT-solver. There are many such theorem provers available. Our choice of theorem prover is discussed in Section 6. To check the validity of a safety property p, we simply prove:

$$\left(\bigwedge_{x=f \in C} x = f \right) \Rightarrow p$$

As is well-known, proving the validity of combinational safety properties is a co-NP-complete problem. Therefore, there are only worst-case exponential time algorithms known that can do this.

Proving Sequential Properties. Next, we show how to check if a safety property p is valid for a given sequential circuit (C, D). Here, we use the method of temporal induction, as described in [9]. The idea is to prove the property in two steps: the base case, and the induction step.

In the base case, we prove that the property holds in the initial state. This corresponds to proving that p is a valid property of the combinational circuit $D_0 \cup (C, D)[1]$. We can use the above method to do that.

In the step case, we prove that if the property holds in a certain state, it will also hold in the next state. This corresponds to proving that $p_1 \Rightarrow p_2$ is valid in the combinational circuit $(C, D)[2]$. This can also be done with the method described above.

Complete Induction. The method of temporal induction as described above is sound, but not complete. This means that there are safety properties which are valid but cannot be proved by the proposed method. In particular, when the safety property p is very weak, assuming it as an induction hypothesis is not enough for the induction step to be a valid formula.

In order to make temporal induction complete, we add the notion of *induction depth*, which is a natural number d. We modify the base case such that it

proves the property for the first d time steps, so that we can assume that the property holds for d steps in the induction step. Thus, the new base case is proving $p_1 \wedge \cdots \wedge p_d$ valid for the circuit $D_0 \cup (C, D)[d]$. The step case becomes proving $p_1 \wedge \cdots \wedge p_d \Rightarrow p_{d+1}$ valid for the circuit $(C, D)[d+1]$.

If a certain property is valid for the base case, but not the step case, we simply increase the induction depth d and try again. If the base case does not hold, we get a trace exhibiting the error back from the theorem prover.

However, it is possible that this process never terminates. This can happen when the unreachable state space contains loops. To exclude the loops and to make the method complete, we can add an extra assumption in the step case, namely that all states used in it are unique. It is sound to assume this, since if there is a solution path leading to a state where p is not true, there also exists a solution path leading to that state which consists of unique states. So, we define $\mathsf{Diff}(D, i, j)$ to mean that the states in time instances i and j are different:

$$\mathsf{Diff}(D, i, j) = \neg \left(\bigwedge_{x:=y \in D} x_i = x_j \right).$$

The new induction step then amounts to proving the following formula valid for the circuit $(C, D)[d+1]$:

$$\left(\bigwedge_{1 \leq i < j \leq d+1} \mathsf{Diff}(D, i, j) \right) \wedge p_1 \wedge \cdots \wedge p_d \Rightarrow p_{d+1}.$$

A more detailed description of temporal induction can be found in [9,2], where the interested reader can find a soundness and completeness proof, and also several ways to make temporal induction stronger.

4 Constructive Semantics of Circuits

In this section, we discuss what circuits mean when we implement them electrically. The interesting case is when the combinational part of a circuit contains a so-called cycle.

Cyclicity. For a given combinational circuit C, and variables x and y, we say that x *directly depends on* y, if y is contained in the right hand side of the definition of x. By taking the transitive closure of this relation, we obtain the relation x *depends on* y.

Now, a combinational circuit C is called *cyclic* if there exists a variable x such

$x = x \wedge x$ (a) $\quad x = \neg x$ (b) $\quad x = x \vee \neg x$ (c) $\quad x = \mathbf{0} \wedge x$ (d)

Fig. 2. Four cyclic circuits

that x depends on itself. A sequential circuit (C, D) is cyclic if C is cyclic.

Electrical Circuits. The circuits we have defined mathematically correspond to an obvious electrical implementation: The logical connectives can be implemented as the corresponding logical gates, points with the same variable name are wired together, and delayed definitions can be implemented using registers. For combinational acyclic circuits, we know that there exists exactly one solution for each input vector. This is also the solution that the corresponding electrical circuit computes.

For cyclic circuits, the classical semantics is too naive. To illustrate this, in Figure 2 we present four examples of cyclic combinational circuits. None of them have any inputs. Circuit (a) has two solutions ($x = \mathbf{0}$ and $x = \mathbf{1}$), circuit (b) has no solutions, and circuit (c) and (d) both have exactly one solution ($x = \mathbf{1}$, resp. $x = \mathbf{0}$).

However, for most existing circuit technologies, such as the standard implementation of gates using CMOS transistors, only circuit (d) electrically computes that solution. Circuit (a), (b) and (c) all lead to undriven outputs.

Electrical Semantics. In order to know what a cyclic circuit means electrically, we have to be able to reason about undriven wires. We do this by extending the boolean domain $\{\mathbf{0}, \mathbf{1}\}$ we have used so far to a ternary domain $\{\bot, \mathbf{0}, \mathbf{1}\}$. We call the value $\bot$ *undefined*, and $\mathbf{0}$ and $\mathbf{1}$ the *defined* values.

In order to talk about solutions of circuits using this new domain, we have to extend our boolean operators to these domains as well. It turns out that the electrical gates in most technologies behave as specified in Figure 3. We can see that, for example, an **and** gate ($\wedge$) produces the output $\mathbf{0}$ as soon as one of its inputs is $\mathbf{0}$, even when the other input is undefined. Also note that the **xor** gate ($\oplus$) is a so-called *strict* operator; as soon as one of its arguments is $\bot$, the output is $\bot$ as well.

If we interpret our new ternary domain as a Scott domain with $\bot$ as the least element, the operators in Figure 3 are called the *parallel extensions* of the corresponding boolean operators. We use the reflexive ordering from the Scott domain in the remainder of this paper; furthermore, $\bot \leq \mathbf{0}$ and $\bot \leq \mathbf{1}$, but $\mathbf{0}$ and $\mathbf{1}$ are incomparable. We can easily see that the operators we are using are all *monotonic* with respect to this ordering. The ordering naturally

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

x	y	$x \wedge y$	$x \vee y$	$x \Rightarrow y$	$x \oplus y$
0	**0**	**0**	**0**	**1**	**0**
0	**1**	**0**	**1**	**1**	**1**
1	**0**	**0**	**1**	**0**	**1**
1	**1**	**1**	**1**	**1**	**0**
$\bot$	**0**	**0**	$\bot$	$\bot$	$\bot$
0	$\bot$	**0**	$\bot$	**1**	$\bot$
$\bot$	**1**	$\bot$	**1**	**1**	$\bot$
1	$\bot$	$\bot$	**1**	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Fig. 3. Parallel extensions of the operators

extends to vectors and sequences of values: $(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n)$ if and only if $x_i \leq y_i$ for all i.

Constructive Combinational Circuits. Extending the boolean domain to a ternary domain possibly increases the number of solutions of a combinational circuit. However, since our operators are monotonic, there always exists a *least* solution. This follows from Tarski's fixed point theorem, as described in [7]. In the paper it is also argued that, given a combinational circuit, and given defined values for all the inputs to the circuit, its electrical counterpart always computes the least solution in the Scott domain!

However, we do not want the circuit to compute undriven values. Thus, we make the following definition. A combinational circuit C is called *constructive*, if and only if, for all defined inputs, all values in the corresponding least solution are defined [10].

We have now established the following. If a circuit is constructive, then we know that its classical semantics corresponds to the constructive semantics, i.e. how the circuit behaves electrically.

Constructive Sequential Circuits. Similarly, we can define what a constructive sequential circuit is. A sequential circuit (C, D) is constructive if, for all k, and for all defined input sequences of length k, all values in the least solution path of length k are defined.

Note that, for non-constructive sequential circuits, least solution paths con-

taining $\perp$ might not faithfully reflect what happens in the circuit electrically. In fact, since it is very much unspecified what happens when a delay component gets a $\perp$ as input, it is difficult to model what happens electrically in non-constructive sequential circuits. But for constructive sequential circuits, our model is accurate.

Weakening. One extra remark can be made here: For a circuit to produce meaningful results, we might want to weaken the definition of constructiveness to only require certain points in the circuit (for example all outputs, and inputs to all registers) to always have defined values, instead of the stronger requirement of all points in the circuit being defined. In the remainder of the paper, we assume the strongest definition of constructiveness, but we can easily adapt the methods to deal with weaker definitions as well.

Unique Solutions. The constructive semantics is defined in terms of least solutions. However, since the domain we are using has a specific shape, we can derive a useful lemma: A combinational circuit C is constructive, if and only if, for all defined inputs, all values in *all* solutions are defined [8]. The proof follows from the fact that, if all solutions are defined, so is the least solution. And if the least solution is defined, there can only be one solution (since the existence of any other defined solution would imply that there is a least solution containing $\perp$), and so all solutions are defined.

We have a similar lemma for sequential circuits. A sequential circuit (C, D) is constructive, if and only if, for all k, and for all defined input sequences of length k, all values in *all* solution paths of length k are defined.

These lemmas are useful, because they eliminate the need for a fixpoint iteration, and thus allow us to formulate the constructiveness condition as a safety property of a circuit.

5 Safety Properties of Cyclic Circuits

In this section, we show how to check safety properties of circuits containing cycles, using the constructive semantics.

Safety and Constructiveness. In order to check a safety property of a circuit with cycles, we should really check that the circuit is constructive first. For example, since circuit (b) in Figure 2 contains a contradiction using the classical semantics, it is possible to prove *any* safety property.

Thus, we make the following definition. Given a circuit, and a safety property p, p is *constructively valid* if the circuit is constructive, and the property p is a

x	(x^0, x^1)
$\mathbf{0}$	$(\mathbf{1}, \mathbf{0})$
$\mathbf{1}$	$(\mathbf{0}, \mathbf{1})$
$\bot$	$(\mathbf{0}, \mathbf{0})$
$?$	$(\mathbf{1}, \mathbf{1})$

Fig. 4. Dual rail encodings of the domains

$$x = y \;\rightarrow\; x^0 = y^0,\; x^1 = y^1 \qquad x = \neg y \;\rightarrow\; x^0 = y^1,\; x^1 = y^0$$

$$x = y \wedge z \;\rightarrow\; x^0 = y^0 \vee z^0,\; x^1 = y^1 \wedge z^1 \qquad x = y \vee z \;\rightarrow\; x^0 = y^0 \wedge z^0,\; x^1 = y^1 \vee z^1$$

$$x = y \Rightarrow z \;\rightarrow\; x^0 = y^1 \wedge z^0,\; x^1 = y^0 \vee z^1 \qquad x = y \oplus z \;\rightarrow\; x^0 = y^0 \wedge z^0 \vee y^1 \wedge z^1,\; x^1 = y^0 \wedge z^1 \vee y^1 \wedge z^0$$

Fig. 5. Dual rail encodings of the combinational definitions

valid safety property for the circuit. So, in order to check if a safety property is valid, we need to check if the circuit is constructive.

Dual Rail Encoding. Constructiveness of both sequential and combinational circuits can be formulated as a safety property in the classical sense, of a transformed version of the circuit. The constructiveness condition is, by the lemma mentioned at the end of the previous section, already almost a safety property. The only difference is the fact that it is formulated over a ternary domain, instead of a boolean domain.

Thus, we encode the ternary domain by introducing two new boolean variables, x^0 and x^1, for every ternary variable x. This is called *dual rail encoding*. In Figure 4 we show how the values of the new variables correspond to the values of x. The variable x^1 is $\mathbf{1}$ whenever x is provably $\mathbf{1}$, and x^0 is $\mathbf{1}$ whenever x is provably $\mathbf{0}$. Note that we have to introduce an extra 'ghost' value $(\mathbf{1}, \mathbf{1})$ in the domain, which does not correspond to any real value. We will ignore solutions containing this ghost value, since it does not correspond to anything in our model.

$$x := y \quad \rightarrow \quad \begin{array}{l} x^0 := y^0 \\ x^1 = \neg x^0 \end{array}$$

Fig. 6. Dual rail encoding of delayed definitions

Definition Encoding. When changing every ternary variable to two binary variables, we have to change the definitions in the circuit accordingly. To do this, we assume that all definitions only contain *shallow* formulas, i.e. formulas containing at most one operator. This can always be obtained by introducing extra variables. Figures 5 and 6 show how to convert such definitions to use the dual rail variables. These definitions correspond to the tables in Figure 3. The property that holds for the dual rail encodings of the operators is that a solution of a definition in the ternary domain corresponds to a classical solution of the transformed definition, according to the dual-rail domain encoding.

Note that every delayed definition is turned into one delayed definition, and one normal definition. This means that the transformation does not increase the number of state variables in a circuit. This is correct since we can assume that the output of a delay component in a circuit is always defined. This assumption is however only valid if we really prove that the input to a delay component is always defined.

The classical solutions of the resulting circuit correspond to the constructive solutions of the original circuit.

Constructiveness as a Safety Property. Now, we formulate the constructiveness condition as a classical safety property: For all defined inputs, all solutions are defined. This is the same as showing that the following property is a valid safety property for the transformed circuit. (We let I be the set of all input variables, and V be the set of all variables in the circuit.)

$$\left(\bigwedge_{i \in I} i^0 \oplus i^1 \right) \Rightarrow \left(\bigwedge_{x \in V} x^0 \vee x^1 \right)$$

We assume that the inputs are defined, using an **xor** ($\oplus$) to assure that (i^0, i^1) is either $(\mathbf{0}, \mathbf{1})$ or $(\mathbf{1}, \mathbf{0})$. Then, we check if all variables are defined, using **or** ($\vee$), to check that they are not $(\mathbf{0}, \mathbf{0})$. The reason we do not use **xor** ($\oplus$) for the other variables also, is because proving the property is easier when weak properties occur on the right hand side of the implication, and strong properties on the left hand side.

Note that this safety property can be used for both combinational and sequential circuits. Thus, to check if a given circuit is constructive, we transform the

circuit as mentioned above, and verify that the above safety property holds for the transformed circuit. We can use any formal verification method for that, provided it gives classical semantics to combinational cycles. For our purposes, the method of temporal induction presented in Section 3 has worked well.

Safety Properties of Cyclic Circuits. As a bonus, instead of first showing constructiveness of a circuit, and later proving the desired safety property, we can do both at the same time. Given that we are interested in a safety property p of a cyclic circuit, we can transform the circuit as above and prove:

$$\left(\bigwedge_{i \in I} i^0 \oplus i^1\right) \Rightarrow \left(\bigwedge_{x \in V} x^0 \vee x^1\right) \wedge p^1$$

This has an advantage especially when doing temporal induction proofs. The success of an inductive proof depends on the stength of the inductive hypothesis. In general, proving multiple properties inductively at the same time can actually reduce the required induction depth. We have observed many cases where the required induction depth of the constructiveness proof went down when proving a strong safety property at the same time.

6 Discussion

Related Work. As mentioned in the introduction, the work that comes closest to ours is the work presented in [8]. There are two main differences between their work and ours: (1) We use a propositional theorem prover and they use decision diagrams; (2) They compute the reachable states of the system whereas we use temporal induction. Our experience with temporal induction is that, for strong enough properties, the required induction depth is very low, so the property becomes relatively easy to prove. Constructiveness is a very strong property, since it expresses information about *all* points in the circuit. In practice, we have found that we usually do not need an induction depth of more than 1 or 2, leading to easy propositional formulas. Note that the induction depth is only dependent on how much the reachability of states affects the constructiveness of the circuit. If the circuit is constructive in all states, even the unreachable ones (or if the circuit is combinational), induction with depth 0 is always enough.

BDDs vs. propositional theorem proving. Both BDDs and propositional theorem proving methods have their own niches in which they work well. BDDs have been used to show constructiveness of circuits with reason-

able success in the past, but we believe that propositional methods are better suited for this job. We believe that the reason for this is that many modern propositional theorem provers are based on algorithms which can globally *propagate* local information in a cheap way. Examples of such algorithms are Stålmarck's method [12] and algorithms that use learning [11]. In many cases, propagation of information is all that is needed to show constructiveness, because often constructiveness only depends on a small part of the control logic of the circuit, and definedness of points is easily propagated to solve the problem. In contrast, when using BDDs, one often has to represent logically irrelevant parts of the circuit too.

Let us take a look at Malik's example, presented in Figure 1 in the introduction). Now, if x is a wide datapath, and correspondingly F and G are complicated functions with large datapaths, any BDD calculation will have difficulties with even representing F and G. However, reasoning about the constructiveness of the circuit is not at all affected by what F and G actually implement; only the number of cycles in the circuit depends on the size of the datapath of F and G. It becomes simply impossible to use BDDs when dealing with this kind of circuit, for example when F and G are multiplier-like circuits [4]. In contrast, though most SAT-solvers also have problems with reasoning *about* multipliers, the constructiveness analysis still only needs to propagate definedness information through F and G, and thus in theory could take linear time. In practice however, our experiments show the time complexity to be closer to quadratic time.

Another example, encountered in one of our experiments, contains a FIFO buffer which is implemented in hardware in the usual way: a sequence of registers connected in a cycle, and a head and tail indexing mechanism. Now, implementing a *find* functionality, which finds the first element in the FIFO satsifying some property within one clock cycle, in the most efficient way possible, requires cycles in the combinational logic. (This is because the search starts at the head and ends at the tail, which are dynamic indexes in a cyclic structure.) Representing FIFOs using BDDs is not practically feasible for large datapaths, since it is well-known that BDDs blow up for this case. However, the constructiveness of the combinational loops is not dependent on the datapath at all. Increasing the datapath up to 2^{32}, our analysis was still capable of proving constructiveness in just a few seconds. Our own BDD-based analyses were incapable of building the BDDs.

To sum up, our main point here is that propagation based theorem proving methods are well-suited for solving this kind of problem.

Implementation. We implemented the proposed analysis in Lava [6], our

synchronous hardware description and verification workbench. Formulating the constructiveness condition as a safety property has the advantage that we can implement the analysis independently from the underlying verification method. In Lava, we just implemented a circuit transformation that performs the translation described in Section 5. The analysis can then be performed by any built-in verification method using classical semantics, such as the ones described in Section 3. The method we mostly use, as described, is induction with increasing depth. We have since extensively used the analysis in our hardware compilation framework [5].

Possible Improvements. One possibility for improvement which we have not pursued is to investigate the circuit structurally, and only apply the analysis on the strongly connected components, i.e. the parts of the circuit involved in a cycle. In practice, this would amount to not applying the dual-rail encoding on the paths that lead from the inputs to the parts of the circuit containing cycles, and not on the paths leading from the cycles to the registers.

Another possibility for efficiency improvement would be to use a SAT-solver that uses a 3-valued logic internally. Many SAT algorithms internally use a structure which would make it easy to adapt them to a 3-valued world.

Limitations and Other Future Work. One limitation of the proposed method is that it does not work when extending the ternary domain with a fourth element, $\top$, representing a *short circuit*, i.e. a point in the circuit driven by both **0** and **1**. As far as we know, computing constructiveness of such circuits requires a fixpoint iteration, because the lemma mentioned at the end of Section 4 does not hold for the corresponding domain. The BDD-based methods in [7,10] still work in this case, because they are capable of computing the least fixed point. Dealing with $\top$ remains a part of our future work.

References

[1] Gérard Berry. The constructive semantics of Pure Esterel. Unfinished draft, available from `http://www.synalp.org`, 1999.

[2] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer Aided Design*, 2000.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[4] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2), 1991.

[5] K. Claessen and G. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits*, 2002.

[6] K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.

[7] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7), July 1994.

[8] K. S. Namjoshi and R. P. Kurshan. Efficient analysis of cyclic definitions. In *Computer Aided Verification*. Springer, 1999.

[9] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, LNCS 1954. Springer, 2000.

[10] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, 1996.

[11] Joao Marques Silva. The GRASP homepage. Available from `http://sat.inesc.pt/~jpms/grasp`, 2000.

[12] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).