Efficient Analysis of Cyclic Definitions

Kedar S. Namjoshi and Robert P. Kurshan

Bell Laboratories
Lucent Technologies
{kedar,k}@research.bell-labs.com
URL : http://cm.bell-labs.com/cm/cs/who/{kedar,k}

Abstract. We present a new algorithm for detecting semantic combinational cycles that is simpler and more efficient than earlier algorithms found in the literature. Combinational circuits with syntactic cycles often arise in processor and bus-based designs. The intention is that external inputs and delay elements such as latches break these cycles, so that no "semantic" cycles remain. Unbroken semantic cycles are considered a design error in this context. Such unbroken cycles may also occur inadvertently in compositions of Mealy machines.
Verification systems that accept semantically cyclic definitions run the risk of certifying systems that have electrically bad or unexpected behavior, while those that prohibit all cyclic definitions constrain the types of systems that can be subjected to formal verification. Earlier work on this issue has led to a reasonable condition, called *Constructivity*, that guarantees the absence of semantic cycles. This formulation is, however, computational in nature, and existing algorithms to decide constructivity are somewhat inefficient. Moreover, they do not apply naturally to circuit definitions in high-level languages that allow variables with non-Boolean types. We propose a new formulation of constructivity, formulated as a satisfiability question, that does not have these limitations. We have implemented the new algorithm in the verification tool COSPAN/FormalCheck. Our experience indicates that the algorithm is simple to implement and usually incurs negligible overhead.

1 Introduction

A circuit may be described as a set of definitions, one for each gate of the circuit. For most circuits, the induced syntactic dependency graph of such a definition is acyclic. Syntactically cyclic definitions, however, occur in many contexts in digital design: Malik [9] points out that it is often desirable to re-use functional units by connecting them in a cyclic fashion through a routing mechanism, and Stok [13] notes that such definitions often arise in the output of synthesis programs. In these cases, the intention is that the routing mechanism can be controlled through external inputs, so that any "semantically" cyclic paths are broken for each valuation of the external "free" inputs and delay elements such as latches. Semantically cyclic definitions may also occur inadvertently in systems composed of several Mealy machines, from feedback connections between the combinational inputs and outputs. Verification systems that accept semantically cyclic definitions run the risk of certifying systems that have behavior that is unexpected or electrically bad, while those that prohibit syntactically cyclic definitions constrain the types of systems that can be subjected to formal verification.

Most current design and verification systems either prohibit all syntactically cyclic definitions, or accept only some of the semantically acyclic definitions. The Esterel compiler is the only existing system we know of that analyzes definitions for semantic cyclicity using the notion of "*Constructivity*" proposed by Berry [2], which considers a circuit to be semantically acyclic iff for every external input, a unique value can be derived for each internal wire by a series of inferences on the definition of the circuit (a precise statement is given in Section 2). Shiple [11] shows that constructive definitions are precisely those that are well-behaved electrically, for any assignment of delay values, in the up-bounded inertial delay model [4].

It is inefficient to check constructivity by enumerating all possible external valuations. Symbolic algorithms for checking constructivity [2, 12, 11] manipulate *sets* of input valuations, representing them with BDD's [3]. This manipulation is based on simultaneous fixpoint equations derived from the circuit definitions and the types of the variables. For variables with k values in their type, these algorithms require k sets of valuations for each variable. Moreover, for arithmetic operations, the fixpoint equations are constructed from partitions (for $+$) or factorizations (for $*$) of all numbers in the type. Thus, these algorithms are somewhat inefficient and difficult to implement for variables with non-Boolean types.

We show in this paper that, by a simple transformation, one can reformulate constructivity as the satisfiability of a set of equations derived from the definitions, over variable types extended with a value $\perp$ (read as "bottom"). This formulation is non-computational and easily extensible to variables with any finite type. The formulation also handles definitions of indexed variables in the same manner. We have implemented this constructivity check in the verification tool COSPAN [7], which is the verification engine for the commercial verification tool FormalCheck; the implementation is simple, and our experience indicates that it usually incurs negligible overhead.

Section 2 motivates and precisely defines constructivity. The new formulation is derived in Section 3. Section 4 describes the implementation of this idea in the COSPAN/FormalCheck verification system. The paper concludes with a discussion of related work and future directions in Section 5.

2 Cyclic Definitions

Notation: The notation generally follows the style in [6]. Function application is represented with a "." and is right-associative; for instance, $f.g.a$ is parsed as $f.(g.a)$. Quantified expressions and those involving associative operators are written in the format $(Q\ x : r.x : g.x)$, where Q is either a quantifier (e.g., $\forall, \exists$) or an associative operator (e.g., $+, *, min, max, lub, glb$), x is the "dummy" variable,

$r.x$ is the range of x, and $g.x$ is the expression. For instance, $(\forall x\ r(x) \Rightarrow g(x))$ is expressed as $(\forall x : r.x : g.x)$, $(\exists x\ r(x) \wedge g(x))$ is expressed as $(\exists x : r.x : g.x)$, and $\sum_{i=0}^{i=n} x_i$ is expressed as $(+i : i \in [0,n] : x.i)$. When the range r is *true* or understood from the context, we drop it and write $(Q\ x :: g.x)$. Proofs are presented as a chain of equivalences or implications, with a hint for each link of the chain. $\Box$

For simplicity, we consider all variables to be defined over a single finite type T. The vocabulary of operator symbols is given by a finite set F. Each symbol in F has an associated "arity", which is a natural number. A symbol f with arity n corresponds to a function $f^* : T^n \rightarrow T$; symbols with arity 0 correspond to values of T. Terms over F and a set of variables X are built as follows : a variable x in X is a term, and for terms $t.i$ $(i \in [0,n))$ and a function symbol f of arity n, $f.(t.0, \ldots, t.(n-1))$ is a term.

Definition 0 (Simultaneous definition). *A* simultaneous definition *is specified by a triple (E, X, Y), where X and Y are disjoint finite sets of variables, E is a set of expressions of the form $y\ ::=\ t$, where $y \in Y$ and t is a term in $X \cup Y$, such that there is exactly one expression in E for each variable in Y.*

In terms of the earlier informal description of a circuit as a set of definitions, X is the set of "external" variables (the free inputs and latches) and Y is the set of "internal" variables (the internal gate outputs); notice that a simultaneous definition contains definitions only for the internal variables. A simultaneous definition induces a *dependency relation* among the variables in Y; for each expression $y\ ::=\ t$, y "depends on" each of the variables appearing in t. A simultaneous definition is *syntactically cyclic* iff this dependency relation contains a cycle. We illustrate some of the subtleties in formulating a correct notion of *semantic acyclicity* with a few examples.

Example 0 : Syntactic Acyclicity

The external variable set is $\{x, y\}$ and the internal variable set is $\{p, q\}$.

$$\begin{aligned} p &::= x \wedge \neg y \\ q &::= x \vee y \end{aligned}$$

This is syntactically acyclic; hence, for every valuation of x and y, p and q have uniquely defined values. $\Box$

Example 1 : Syntactic Cyclicity, Semantic Acyclicity

The external variable set is $\{x, y\}$ and the internal variable set is $\{p, q\}$.

$$\begin{aligned} p &::= \underline{\textsf{if}}\ x\ \underline{\textsf{then}}\ y\ \underline{\textsf{else}}\ q \\ q &::= \underline{\textsf{if}}\ x\ \underline{\textsf{then}}\ p\ \underline{\textsf{else}}\ x \end{aligned}$$

This is syntactically cyclic; however, notice that if x is *true*, the definition simplifies to the acyclic definition:

$$\begin{aligned} p &::= y \\ q &::= p \end{aligned}$$

Similarly, the simplified definition is acyclic when x is *false*. Thus, each setting of the external variable x breaks syntactic cycles. □

Example 2 : Semantic Cyclicity

The external variable set is $\{x\}$ and the internal variable set is $\{p, q\}$.

$$p ::= q \wedge x$$
$$q ::= p$$

This is syntactically cyclic. If x is *false*, the simplified definition is acyclic; however, when x is *true*, it simplifies to one that presents a semantic cycle:

$$p ::= q$$
$$q ::= p$$

□

A plausible semantics for a simultaneous definition is to interpret each expression $y ::= t$ as an equation $y = t$, and declare the definition to be semantically acyclic if this set of simultaneous equations has a solution for each valuation of the external variables. With this semantics, Examples 0 and 1 are semantically acyclic, but so is Example 2. One may attempt to rectify this situation by requiring there to be a *unique* solution for each input valuation; the following example illustrates that this is also incorrect.

Example 3: Incorrectness of the "unique solution" criterion.

The external variable set is $\{x\}$ and the internal variable set is $\{p, q\}$.

$$p ::= p \wedge x$$
$$q ::= \underline{\mathsf{if}}\ p\ \underline{\mathsf{then}}\ \neg q\ \underline{\mathsf{else}}\ \mathit{false}$$

This is syntactically cyclic. If x is *false*, the simplified definition is acyclic, and hence has a unique solution. If x is *true*, the simplified definition is the following.

$$p ::= p$$
$$q ::= \underline{\mathsf{if}}\ p\ \underline{\mathsf{then}}\ \neg q\ \underline{\mathsf{else}}\ \mathit{false}$$

This has the unique solution $p = \mathit{false}, q = \mathit{false}$. Hence, the definition has a unique solution for each valuation of x ! The "unique solution" criterion thus leaves the cycles $p ::= p$, $q ::= \neg q$ undetected. □

The examples suggest that a straightforward formulation in terms of solutions to the simultaneous equations may not exist. Berry [2], strengthening a formulation of Malik [9], proposed a condition called *Constructivity.* Constructivity is based on the simplification process that was carried out informally in the examples above : for each valuation of the external variables, one attempts to simplify the right hand sides of the definitions. If a term t in a definition $y ::= t$ simplifies to a constant a, the current valuation is extended with $y = a$, and the definition $y ::= t$ is removed. The simplifications are restricted to cases where the result is defined by the current valuation irrespective of the values of variables that are currently undefined. For instance, with $\{x = \mathit{false}\}$ as the current valuation, $\underline{\mathsf{if}}\ x\ \underline{\mathsf{then}}\ y\ \underline{\mathsf{else}}\ z$ simplifies to z; $x \wedge y$ simplifies to *false*; but $y \vee \neg y$ *does not* simplify to *true*. Berry [2] shows that this process produces a unique result, *independent* of the order in which simplification steps are applied. The appropriateness of constructivity is shown by Shiple [11], who demonstrates that constructive definitions are precisely those that are well-behaved electrically, for any assignment of delay values, in the up-bounded inertial delay model [4]. Malik [9] shows that the problem of detecting semantic cyclicity is NP-complete.

Definition 1 (Constructivity). *A simultaneous definition is semantically acyclic iff for each valuation of the external variables, the simplification process leads to an empty set of definitions.*

3 Constructivity as Satisfiability

There is another way of viewing the simplification process that leads to our new formulation. Simplification is seen as a fixpoint process that computes the "maximal" extension of the original valuation of external variables (maximal in the sense that the set of definitions cannot be simplified further with this valuation). The algorithms for checking constructivity proposed in [9, 12] use this fixpoint formulation. We show (Theorem 1 below) that it is possible to recast the fixpoint formulation as a satisfiability question. This observation lets us develop a simple algorithm for constructivity that extends easily to non-Boolean types.

3.1 Background

To formulate simplification as a fixpoint process, we need some well-known concepts from Scott's theory of Complete Partial Orders (CPO's) [10]. The type T is extended with a new element $\perp$ (read as "bottom") to form the type $T_\perp$. $T_\perp$ is equipped with the partial order $\preceq$, defined by $a \preceq b$ iff $a = b$ or $a = \perp$. Note that $\preceq$ is a CPO (every sequence of elements that is monotonically increasing w.r.t. $\preceq$ has a least upper bound). The greatest lower bound (glb) of two elements a, b is defined as : $glb.(a, b) = \mathit{if}\ a \neq b\ \mathit{then}\ \perp\ \mathit{else}\ a$. The ordering $\preceq$ is extended point-wise to vectors on $T_\perp$ by $u \sqsubseteq v$ iff $|u| = |v| \ \wedge\ (\forall i :: u.i \preceq v.i)$. This ordering is a CPO on the set of vectors on $T_\perp$. The greatest lower bound is also defined point-wise over vectors of the same length: $glb.(u, v) = w$, where for every i, $w.i = glb.(u.i, v.i)$.

For each function symbol f in F, $f_\perp$ is a symbol of the same arity that indicates application to $T_\perp$ rather than to T. The interpretation $f_\perp^*$ of $f_\perp$ over $T_\perp$ should be a function that extends f^* and is monotone w.r.t. the order $\sqsubseteq$; i.e., for vectors u, v of length the arity of $f_\perp$, $u \sqsubseteq v$ implies $f_\perp^*.u \sqsubseteq f_\perp^*.v$. The ordering $\sqsubseteq$ and the monotonicity condition encodes the informal description of $\perp$ as the "undefined" value: if v is "more defined" than u, then $f_\perp^*.v$ should also be "more defined" than $f_\perp^*.u$. The extension of a term t is represented by $t_\perp$ and is defined recursively based on the structure of the term: $(x)_\perp = x$; $(f.(t.0, \ldots, t.(n-1)))_\perp = f_\perp.(t_\perp.0, \ldots, t_\perp.(n-1))$. It is straightforward to show that the interpretation of an extended term is also monotonic w.r.t. $\sqsubseteq$. Every monotonic function on a CPO has a least fixpoint.

3.2 Constructivity as a Fixpoint Process

A partial valuation constructed during the simplification process can now be represented as total function from $X \cup Y$ to $T_\perp$, where currently undefined variables are given the value $\perp$. An initial valuation V is a function that maps X into T and Y to $\{\perp\}$. At each step, for some non-deterministically chosen definition $y ::= t$, the current valuation V is updated to $V.[y \leftarrow t^*_\perp.V]$. By an argument [1] (cf. [5]) based on monotonicity, this non-deterministic process terminates with a valuation that is the simultaneous least fixpoint of the derived set of equations $\{y = t^*_\perp \mid (y ::= t) \in E\}$. For a simultaneous definition $C = (E, X, Y)$, let $(\mathit{lfp}\ Y : E^*.(X, Y))$ denote this least fixpoint. The fixpoint depends on, and is defined for, each valuation of X. The constructivity definition can now be re-stated as follows.

Definition 2 (Constructivity-FIX). *A simultaneous definition* (E, X, Y) *is semantically acyclic iff for each initial valuation* V*, the vector* $(\mathit{lfp}\ Y : E^*.(V, Y))$ *has no* $\perp$*-components.*

For a vector v over $T_\perp$, let $\perp\!\mathit{free}.v$ be the predicate $(\forall\ i :: v.i \neq \perp)$. The constructivity condition is precisely $(\forall\ v : \perp\!\mathit{free}.v : \perp\!\mathit{free}.(\mathit{lfp}\ Y : E^*.(v, Y)))$. Malik [9] checks a weaker condition in which the set of internal variables Y has a subset of "output" variables W. Let $\mathit{output}\!\perp\!\mathit{free}.v$ be the predicate $(\forall\ i : i \in W : v.i \neq \perp)$. Malik's condition can be phrased as : $(\forall\ v : \perp\!\mathit{free}.v : \mathit{output}\!\perp\!\mathit{free}.(\mathit{lfp}\ Y : E^*.(v, Y)))$.

Checking the Constructivity-FIX condition independently for each initial valuation is inefficient. Malik, Berry, Touati and Shiple [9, 2, 12, 11] use a derived scheme that operates on sets of external valuations. If the type T has k elements, the scheme associates k subsets with each variable y in Y: the set $y.i$, $i \in [0, k)$, contains external valuations for which the variable y evaluates to i. These subsets are updated by set operations derived from the semantics of the basic operators. For instance, for the definition "$x ::= y \wedge z$", the updates are given by $x.\mathit{false} = y.\mathit{false} \cup z.\mathit{false}$, and $x.\mathit{true} = y.\mathit{true} \cap z.\mathit{true}$.

This scheme has two limitations that arise for non-Boolean types: (i) the algorithm has to maintain k sets for each variable, and (ii) the set operations needed can be quite complex when the basic operators include (bounded) arithmetic. For example, for the definition $x ::= y + z$, $x.k$ would be defined as $y.l + z.m$, for various partitions of k as $l + m$; similarly, for $x ::= y * z$, $x.k$ would be defined as $y.l * z.m$, for various factorizations of k as $l * m$. Our new formulation, Constructivity-SAT, changes Constructivity-FIX to a satisfiability question and avoids these difficulties.

3.3 Constructivity as Satisfiability

The new formulation (apparently) strengthens the Constructivity-FIX definition to require that *every* fixpoint of E^* is $\perp$-free. The equivalence of the two formulations is shown in Theorem 1.

Definition 3 (Constructivity-SAT). *A simultaneous definition* (E, X, Y) *is semantically acyclic iff* $(\forall\, v, u : \bot\mathit{free}.v \;\wedge\; u = E^*.(v, u) : \bot\mathit{free}.u)$.

Lemma 0. *For a monotone property* P *and a monotone function* f *on a CPO* $\sqsubseteq$, $P.(\mathit{lfp}\ X : f.X)$ *iff* $(\forall\, u : u = f.u : P.u)$.

Proof. The implication from right to left is trivially true, as $(\mathit{lfp}\ X : f.X)$ satisfies the condition $u = f.u$. For the other direction, note that the fixpoints of f are partially ordered by $\sqsubseteq$, with the least fixpoint below any other fixpoint. By the monotonicity of P, if P holds of the least fixpoint, it holds of every fixpoint. □

Theorem 1. *Constructivity-FIX and Constructivity-SAT are equivalent.*

Proof. For any simultaneous definition $C = (E, X, Y)$,

$$
\begin{array}{ll}
 & C \text{ satisfies Constructivity-FIX} \\
\equiv & \quad \{ \text{ by definition } \} \\
 & (\forall\, v : \bot\mathit{free}.v : \bot\mathit{free}.(\mathit{lfp}\ Y :\ E^*.(v,\, Y))) \\
\equiv & \quad \{ \bot\mathit{free} \text{ is monotone w.r.t. } \sqsubseteq;\ \text{Lemma 0 } \} \\
 & (\forall\, v : \bot\mathit{free}.v : (\forall\, u : u = E^*.(v, u) : \bot\mathit{free}.u)) \\
\equiv & \quad \{ \text{ rearranging } \} \\
 & (\forall\, v, u : \bot\mathit{free}.v \;\wedge\; u = E^*.(v, u) : \bot\mathit{free}.u) \\
\equiv & \quad \{ \text{ by definition } \} \\
 & C \text{ satisfies Constructivity-SAT}
\end{array}
$$

□

The extension of a function f from T^n to $T_\bot^n$ can be defined in general as follows: the value of the extension at a vector v is the greatest lower bound of the function values at $\bot$-free vectors above v in the order. Formally, $f_\bot^*.v = (\mathit{glb}\ w : \bot\mathit{free}.w \;\wedge\; v \sqsubseteq w : f^*.w)$. It is straightforward to show that this is a monotone extension of f. The extensions of basic arithmetic and Boolean functions are easily determined by this formula. For example, the extension of $\wedge$ is given by:

$$
u \wedge_\bot v = \begin{array}{lll}
\mathit{false} & \text{if } u = \mathit{false} \text{ or } v = \mathit{false}; & \text{otherwise,} \\
\bot & \text{if } u = \bot \text{ or } v = \bot; & \text{otherwise,} \\
u \wedge v & &
\end{array}
$$

To illustrate the use of the general formulation, we can check that

$$
\begin{array}{ll}
 & u \wedge_\bot \mathit{false} \\
= & \quad \{ \text{ by the general formulation } \} \\
 & (\mathit{glb}\ x, y : x \neq \bot \;\wedge\; y \neq \bot \;\wedge\; u \preceq x \;\wedge\; \mathit{false} \preceq y : x \wedge y) \\
= & \quad \{ \text{ definition of } \preceq \ \} \\
 & (\mathit{glb}\ x : x \neq \bot \;\wedge\; u \preceq x : x \wedge \mathit{false}) \\
= & \quad \{ \text{ definition of } \wedge \ \} \\
 & (\mathit{glb}\ x : x \neq \bot \;\wedge\; u \preceq x : \mathit{false}) \\
= & \quad \{ \text{ definition of } \mathit{glb} \ \} \\
 & \mathit{false}
\end{array}
$$

The extension of $*$ is similar to that for $\wedge$, with 0 substituted for *false*. The extension of $+$ is given below :

$$u +_\perp v = \begin{cases} \perp & \text{if } u = \perp \text{ or } v = \perp; \\ u + v & \text{otherwise,} \end{cases}$$

The extensions of other basic operators can be defined equally easily. The new formulation thus overcomes both the limitations of the earlier one: the extensions are easy to define and compute, and we do not need to maintain sets of valuations for each variable; the only changes required are to extend both the types of variables and the definitions of the basic operators.

3.4 Indexed Variables

In many input languages, including the S/R language of the COSPAN system, it is possible to declare arrays of variables. If z is such an array variable, definitions of the form $z[c] \ ::= \ t$, where c is a constant, can be handled with the machinery presented earlier, by treating $z[c]$ as an ordinary variable. A definition of the form $z[e] \ ::= \ t$, however, where e is a non-constant term, cannot be handled with the earlier machinery, as it corresponds to the set of definitions $\{z[c] ::= \underline{\mathsf{if}}\ (e = c)\ \underline{\mathsf{then}}\ t \mid c \in \mathit{indices}(z)\}$. Notice that the term $\underline{\mathsf{if}}\ (e = c)\ \underline{\mathsf{then}}\ t$ is a *partial* function. As a typical instance, consider the following definition, where z is an array indexed by $\{0, 1\}$, and x and y are variables.

$$\begin{aligned} z[x] &::= a \\ z[y] &::= b \end{aligned}$$

The semantics of S/R requires that the valuations of x and y be distinct. The defining term for $z[0]$ is the partial function *if* $x = 0$ *then* a *else if* $y = 0$ *then* b. This term may itself be considered as a partial function on $T_\perp$, defined only for $x = 0$ and $y = 0$. With this interpretation, it is monotonic [1] w.r.t. $\sqsubseteq$. Recombining the terms for $z[0]$ and $z[1]$, one obtains the following modification (for $T_\perp$) of the original definitions for $z[x]$ and $z[y]$:

$$\begin{aligned} z[x] &::= \underline{\mathsf{if}}\ x \neq \perp\ \underline{\mathsf{then}}\ a \\ z[y] &::= \underline{\mathsf{if}}\ y \neq \perp\ \underline{\mathsf{then}}\ b \end{aligned}$$

These definitions contribute in the following way to the induced "equations":

$$\begin{aligned} &(x \neq \perp) \Rightarrow (z[x] = a) \\ &(y \neq \perp) \Rightarrow (z[y] = b) \end{aligned}$$

4 Implementation

We have implemented this new formulation in the COSPAN/FormalCheck verification system [7]. The input language for the COSPAN system is S/R ("selection/resolution") [8]. An S/R program consists of a number of processes,

[1] A partial function f is monotonic w.r.t. a partial order $\preceq$ iff whenever $x \preceq y$ and f is defined at x, f is defined at y and $f.x \preceq f.y$.

which may be viewed as Mealy machines with Rabin/Streett-type acceptance conditions. The variables of each process are either state or selection variables. Selection variables, in turn, are either free (unconstrained) inputs or combinational variables used to determine the next-state relation of the system [8]. In the terminology of the earlier sections, the state variables together with the free input variables form the "external" variables, since the inputs and state variables do not change value for the duration of the selection cycle; the other selection variables form the "internal variables".

There are no restrictions in S/R on the dependencies between selection variables: selection variables declared within a process may be mutually interdependent and may be used as inputs to other processes, thus potentially introducing syntactic cycles that span process boundaries. In addition, the presence of *semantic* cycles may depend on the valuation of the state variables. For instance, a semantic cycle may be "unreachable", if the particular states in which it is induced are unreachable. The question, then, is to identify whether any semantic cycles are present in reachable states of the program for some free-input valuation (this problem is shown to be PSPACE-complete in [11]).

The S/R compiler parses the program and analyzes syntactic dependencies among internal variables. If there is a syntactic cycle, it identifies a set of internal variables whose elimination would break each syntactic cycle; such a set is commonly called a "feedback vertex set". The parser retains the variables in the feedback vertex set, and macro-expands the other variables, so that the variables in the feedback vertex set are defined in terms of themselves and the input and state variables. In the terminology used earlier, these remaining internal variables and their defining terms form the simultaneous definition that is to be analyzed. We will refer to these internal variables as the "relevant" variables. Each relevant variable is treated as a state variable for reachability analysis.

Our implementation uses a single MTBDD terminal to represent $\perp$. While MTBDD's for multiplication are exponential in the number of bits, they represent most other operations efficiently and are therefore used in COSPAN. The types of the relevant variables are extended to include the $\perp$-terminal. The types of input and state variables are not extended. The implementation includes a library of extended basic operators, defined as described in Section 2. These extend the basic operators of S/R, including Boolean operators, arithmetic operators such as $+, *, div, exp, mod$, and conditional operators such as if then else .

Each definition $x ::= t$ of a relevant non-indexed variable is converted to the equation $x = t^*_\perp$, while a definition $z[e] ::= t$ of an indexed variable is converted to $(e^*_\perp \neq \perp) \Rightarrow (z[e^*_\perp] = t^*_\perp)$, as described in Section 3.4. The conjunction of these formulae forms the simultaneous fixpoint term $Y = E^*.(S, X, Y)$, where S is the set of state variables, X is the set of free input variables, and Y is the set of relevant variables. The Constructivity-SAT formula determines (by negation) the following predicate on state variables:

$$Cyclic.S = (\exists X, Y : Y = E^*.(S, X, Y) \ \wedge \ \neg\perp free.Y).$$

The predicate $\neg Cyclic$ is checked for invariance during reachability analysis; if it fails, the system automatically generates an error-track leading from an initial state to a state s such that $Cyclic.s$ is true. It is not difficult to recover the set of variables involved in a semantic cycle for a particular input k at state s by inspecting the BDD for $(Y = E^*.(s,k,Y) \;\wedge\; \neg\bot \mathit{free}.Y)$ – every path to a 1-node includes variables that have value $\bot$; these variables are involved in a semantic cycle.

The description above should indicate that implementing the constructivity check with the new formulation is a fairly simple process. We have experimented with this implementation on a test suite for COSPAN formed of several large programs that represent real designs. While syntactic cycles are usually short, some of our examples had cycles of length greater than 20. Our experience has been that, in most cases, the run-time and BDD sizes increase, if at all, by a negligible amount. There are a few cases where the BDD sizes increase by a large amount, and even some where the sizes decrease – this seems to be attributable to the irregular behavior of the dynamic reordering algorithms. We have not conducted a thorough comparison with the algorithm in [12], but it is reasonable to expect that our algorithm will be more efficient for non-Boolean variables, as it avoids both the large number of BDD's and the fixpoint computation. It is less certain whether our algorithm offers a large improvement on the earlier one in the case when all variables are Boolean; this would require experimental comparison. In any case, the potential benefits of detecting semantic cycles before circuit fabrication far outweigh the disadvantage of the (usually small) time and memory increases that we have observed for our detection process.

5 Related Work and Conclusions

The work most related to ours is by Berry [2] and Shiple [11]. Berry proposed the original operational formulation of constructivity (Constructivity) and the denotational formulation (Constructivity-FIX), based on work by Malik [9]. These definitions are based on computational processes – one would prefer a non-computational definition of the concept of "semantic acyclicity". Shiple, Berry and Touati [12, 2, 11] and Malik [9] propose symbolic, fixpoint-based algorithms to check constructivity. These algorithms are difficult to implement and somewhat inefficient for variables with non-Boolean types.

Our new formulation overcomes both limitations, by presenting a simple, non-computational definition of constructivity (Constructivity-SAT) and a symbolic algorithm based on the new formulation that is simple to implement for variables with arbitrary finite types. Our initial experiments with the implementation of this algorithm in the formal verification system COSPAN/FormalCheck indicate that in most cases it has minimal, if any, adverse impact on the execution time and BDD sizes. It should be quite easy to incorporate this algorithm into other verification and synthesis tools. As in [11], one can also determine the set of input values for which a circuit is constructive, by not quantifying over v in the Constructivity-SAT definition.

In [11] Shiple considers a class of cyclic combinational circuits whose behavior is based on the assumption that the circuit retains "state" across clock cycles (for example, a flip-flop implemented by a pair of cross-connected NAND gates). It would be interesting to see if our formulation of constructivity can be modified to analyze such sequential behavior.

Acknowledgements: Thanks to Tom Szymanski for providing references to work on constructivity, and to Kousha Etessami, Mihalis Yannakakis, and Jon Riecke for useful comments and discussions about this work.

References

[1] H. Bekič. Definable operations in general algebras, and the theory of automata and flowcharts. Technical report, IBM, 1969. Reprinted in Programming Languages and Their Definition, LNCS 177, 1984.

[2] G. Berry. *The Constructive Semantics of Esterel.* Draft book, available at ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness.ps.gz, 1995.

[3] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[4] J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1994.

[5] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice (rapport de recherche r.r. 88). Technical report, Laboratoire IMAG, Universite' scientifique et me'dicale de Grenoble, 1978.

[6] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[7] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. CAV'96*, volume 1102, pages 423–427. LNCS, 1996.

[8] J. Katzenelson and R. P. Kurshan. S/R: A language for specifying protocols and other coordinating processes. In *Proc. IEEE Conf. Comput. Comm.*, pages 286–292, 1986.

[9] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design*, 1994.

[10] D. S. Scott. A type-theoretical alternative to CUCH, ISWIM, OWHY. Unpublished notes, Oxford, 1969. Published in *Theoretical Computer Science*, 1993.

[11] T. Shiple. *Formal Analysis of Synchronous Circuits.* PhD thesis, Univerisity of California, Berkeley, 1996.

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

[13] L. Stok. False loops through resource sharing. In *International Conference on Computer-Aided Design*, 1992.

6 Appendix

This appendix contains the definitions of the extended basic operators of S/R.

Boolean Operators:

$$
\begin{array}{llll}
u \wedge_\perp v = & \mathit{false} & \text{if } u = \mathit{false} \text{ or } v = \mathit{false}; & \text{otherwise,} \\
 & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u \wedge v & & \\
u \vee_\perp v = & \mathit{true} & \text{if } u = \mathit{true} \text{ or } v = \mathit{true}; & \text{otherwise,} \\
 & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u \vee v & & \\
\neg_\perp u = & \perp & \text{if } u = \perp; & \text{otherwise,} \\
 & \neg u & &
\end{array}
$$

Arithmetic Operators:

$$
\begin{array}{llll}
u +_\perp v = & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u + v & & \\
u *_\perp v = & 0 & \text{if } u = 0 \text{ or } v = 0; & \text{otherwise,} \\
 & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u * v & & \\
u \; \mathit{div}_\perp \; v = & 0 & \text{if } u = 0; & \text{otherwise,} \\
 & \perp & \text{if } (u = \perp \text{ and } v \neq 0) \text{ or } v = \perp; & \text{otherwise,} \\
 & u \; \mathit{div} \; v & & \\
u \; \mathit{mod}_\perp \; v = & 0 & \text{if } u = 0 \text{ or } v = 1; & \text{otherwise,} \\
 & \perp & \text{if } (u = \perp \text{ and } v \neq 0) \text{ or } v = \perp; & \text{otherwise,} \\
 & u \; \mathit{mod} \; v & & \\
u \; \mathit{exp}_\perp \; v = & 0 & \text{if } u = 0; & \text{otherwise,} \\
 & 1 & \text{if } u = 1 \text{ or } v = 0; & \text{otherwise,} \\
 & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u \; \mathit{exp} \; v & &
\end{array}
$$

Comparison Operators:

$$
\begin{array}{llll}
u <_\perp v = & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u < v & & \\
u \leq_\perp v = & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u \leq v & & \\
u =_\perp v = & \perp & \text{if } u = \perp \text{ or } v = \perp; & \text{otherwise,} \\
 & u = v & &
\end{array}
$$

Conditional Operators:

$$
\begin{array}{llll}
(\underline{\mathsf{if}} \; c \; \underline{\mathsf{then}} \; u \; \underline{\mathsf{else}} \; v)_\perp = & u & \text{if } c = \mathit{true}; & \text{otherwise,} \\
 & v & \text{if } c = \mathit{false}; & \text{otherwise,} \\
 & u & \text{if } c = \perp \text{ and } u = v; & \text{otherwise,} \\
 & \perp & \text{if } c = \perp \text{ and } u \neq v & \\
(\underline{\mathsf{if}} \; c \; \underline{\mathsf{then}} \; u)_\perp = & u & \text{if } c = \mathit{true} &
\end{array}
$$