Constructive Analysis of Cyclic Circuits

Thomas R. Shiple
Department of EECS
University of California, Berkeley

Gérard Berry
Ecole des Mines
Sophia-Antipolis

Hervé Touati
Digital Equipment Corporation
Paris

Abstract

Traditionally, circuits with combinational loops are found only in asynchronous designs. However, combinational loops can also be useful for synchronous circuit design. Combinational loops can arise from high-level language behavioral compiling, and can be used to reduce circuit size. We provide a symbolic algorithm that detects if a sequential circuit with combinational loops exhibits standard synchronous behavior, and if so, produces an equivalent circuit without combinational loops. We present applications to hardware and software synthesis from the Esterel synchronous programming language.

1 Introduction

The interpretation of a loop-free combinational circuit[1] is clear. On the logical side, the circuit defines a boolean function f from inputs to outputs; the function f is computed by composing the intermediate functions according to the gate operators. On the electrical side, if the input voltages remain stable at logical values, then after some delay, the output voltages stabilize to the unique logical values defined by f. Classical synchronous circuit design is based on this perfect match between the electrical input-output correspondence and the boolean function f.

The situation is much less clear when the combinational part of a circuit, meant to be synchronous, has loops. It is easy to build combinational circuits that cannot be described by boolean functions, and do not stabilize electrically to unique logical values. Prototypical examples are the circuits C_1 defined by $x = x$ and C_2 defined by $x = \overline{x}$ where x is an output[2]; C_1 has two boolean solutions, while C_2 has none. Since cycles give rise to circuits with multiple solutions or no solutions, they are usually avoided in synchronous circuit design.

However, cyclic combinational circuits may have well-defined logical and electrical behaviors, with the same perfect match as for acyclic ones. Such circuits can be safely used in synchronous designs; they appear, for example, as the result of synthesis from synchronous programs written in the Esterel language [1]. This paper presents a symbolic procedure to analyze the behavior of cyclic circuits. This procedure is currently used in the Esterel v4 compiler. Our work builds upon Malik's [11] work, extending his analysis method to sequential circuits, and providing a refined fixed point iteration method.

[1]We refer to an arbitrary interconnection of logic gates as a combinational circuit.

[2]We denote negation, disjunction, and conjunction by $\overline{x}$, $x + y$, and $x \cdot y$, respectively.

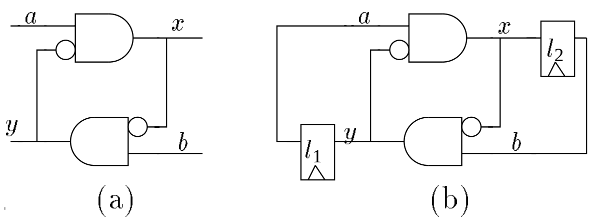

Figure 1: Circuits are well-behaved unless $a = b = 1$.

1.1 Well-behaved cyclic circuits

Consider the circuit C_3, shown in Figure 1a, with inputs a, b and outputs x, y defined by the equations:

$$\begin{aligned} x &= a \cdot \overline{y} \\ y &= b \cdot \overline{x}. \end{aligned}$$

The logical behavior is well-defined unless $a = b = 1$. If $a = 0$, then we have $x = 0$ and $y = b$; symmetrically, if $b = 0$, then $y = 0$ and $x = a$. However, if $a = b = 1$, then x and y are undefined since their equations become $x = \overline{y}$ and $y = \overline{x}$. Similarly, the electrical behavior is well-defined if both inputs are not 1: the outputs of the circuit stabilize for all gate and wire delays. Altogether, we can say that the circuit is *partially defined*, and that under the assumption $a \cdot b = 0$, it behaves like the acyclic circuit C_4 defined by $x = a$ and $y = b$. Our procedure reports under which inputs C_3 is well-defined, and produces the acyclic version C_4.

The above analysis extends to sequential circuits. Suppose two latches are added to C_3 (see Figure 1b), where l_1 is initialized to 1 and l_2 to 0. In this case, the behavior is always well-defined, since the unstable state $a = b = 1$ is never reached. Using state reachability analysis, our procedure confirms this, and thus declares this circuit well-behaved.

This example is representative of cyclic circuits generated by high-level synthesis from languages such as Esterel. For such languages, it is essential to detect whether cyclic circuits have well-defined behavior, so that well-behaved circuits are not rejected. In addition, Malik presents another class of examples where benign cycles result from sharing subcircuits in order to reduce circuit size. Such cycles can be generated by behavioral compilers from languages such as VHDL.

1.2 Constructive circuits

We have to be more precise about what we call a well-behaved circuit. Consider first circuits without latches.

For each input vector in a given input care set, we want the circuit to have 1) a unique boolean solution for each output, and 2) electrical stabilization of the outputs for all possible wire and gate delays. For circuits with latches, we want these properties to hold for all input sequences in a given care set.

Surprisingly, these two notions are not identical. Consider the (inputless) circuit C_5 with output x defined by $x = x + \overline{x}$. It so happens that (1) holds since $x = 1$ is the unique boolean solution, but that (2) does not hold since x does not stabilize for certain delay values. This circuit is not well-behaved, and illustrates that we do not accept a circuit just because it has a unique boolean solution. Furthermore, it illustrates that two circuits (for example C_5 and the constant circuit $x = 1$) may compute the same boolean function, but one is well-behaved and the other is not. Thus, well-behavedness is a structural property.

To capture the intuition of well-behaved circuits described above, we introduce the class of *constructive circuits*. Constructive circuits can be characterized in three different ways:

1. *The logical definition,* which is based on a constructive version of boolean logic. In this logic, $x + y = 1$ is provable from either $x = 1$ or $y = 1$, and $x + y = 0$ is provable from $x = y = 0$. The law of excluded middle, $x + \overline{x} = 1$, does not hold, unless x has been proved to be either 0 or 1 (this explains why circuit C_5 is rejected).

2. *The semantical definition,* which is based on Scott booleans $\{0, 1, \perp\}$, as in classical denotational semantics [7, 12]. A circuit is thought of as a monotonic function on the lattice $\{0, 1, \perp\}$ whose least fixed point defines the output. For a circuit that is monotonic, changing an input from $\perp$ to 1 or 0 cannot cause an output to change from 1 or 0 to $\perp$. In C_5, the least fixed point is $x = \perp$ and the circuit is undefined. (This is the same as ternary circuit analysis using values $\{0, 1, X\}$ [4].)

3. *The electrical definition,* which uses the inertial delay model of [4] and requires electrical stabilization for all delays.

The exact definitions are given in [2], where it is proved that they are indeed equivalent.

The contribution of this paper is a practical algorithm that can determine whether a circuit with latches is constructive. The algorithm performs the analysis of a circuit relative to care inputs and reachable states. If a circuit is constructive, then an equivalent circuit without combinational cycles is generated; otherwise, an error trace is reported giving a sequence of inputs in the given care set leading to a state where the output is not constructively determined. This problem was motivated by the occurrence of cyclic Esterel programs. Our algorithm has found a significant application in the new Esterel compiler by admitting circuits that are well-behaved according to the Esterel semantics, but which were rejected by the previous Esterel compiler.

2 Related work

We are not aware of any other procedure that determines whether a cyclic circuit with latches is constructive. However, Malik provided a procedure for circuits without latches, and our work builds on this (see Section 3.1).

Burch et al. [5] use ternary valued relations to model combinational loops. Their model allows the description of "apparent loops" (loops between hierarchical blocks that in fact disappear when the circuit is flattened) and "actual loops." However, they do not address the problem of deciding if a circuit is constructive.

Brzozowski and Seger [4] address the following problem: given that a circuit (without latches) is in a logically consistent state, what is the effect of changing the logical values on a subset of inputs? They solve this problem using ternary valued simulation. Although their work is primarily focused on asynchronous circuits, it is the key for proving the equivalence between our three definitions of constructive circuits.

Halbwachs et al. [9] give a procedure that detects whether a circuit has a unique boolean solution. Their test accepts circuit C_5 above because they do not require electrical stabilization; hence, their classification of circuits is different than ours. However, like us, they handle circuits with latches and input care sets, and for circuits with unique boolean solutions, they can generate equivalent acyclic circuits.

3 Analysis of circuits without latches

In this section we discuss the constructive analysis of circuits composed of arbitrary interconnections of boolean gates.

3.1 Malik's procedure

Malik uses ternary valued symbolic simulation to decide if a circuit is constructive. Central to ternary valued simulation is the concept of the *ternary valued extension* of a boolean function. The following tables show the ternary valued extension for several boolean operators.[3]

a	NOT
0	1
1	0
X	X

a	b	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
X	0	0	X	X
X	1	X	1	X
0	X	0	X	X
1	X	X	1	X
X	X	X	X	X

These are called the "parallel" extensions in denotational semantics [12]. They follow the basic rule, which is valid in the electrical model, that a 0 or 1 output value can be deduced whenever there is sufficient information available at the inputs. For example, a 0 at any input of an AND gate forces the output to 0. An important fact is that all of these extensions are monotonic.

Malik's procedure works by computing the ternary valued function, in terms of the primary circuit inputs, for each node in the circuit. Since the primary inputs are assumed to be known, binary valued signals, only functions of the form $f : \{0, 1\}^n \rightarrow \{0, 1, X\}$ need to represented; we refer to such functions as *ternary valued functions* (TVFs). A pair of boolean functions (f^1, f^0) is used to encode a TVF f,

[3]We use the symbol X for the undefined (or uncertain) value instead of $\perp$, since X is more common in circuit analysis.

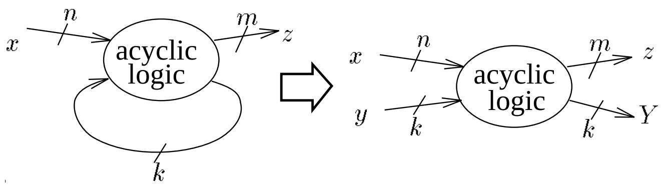

Figure 2: Breaking feedback arcs to produce an acyclic circuit.

such that f^1 (resp. f^0) is the characteristic function of the set of inputs for which f evaluates to 1 (resp. 0). f^1 and f^0 are represented by binary decision diagrams (BDDs). We refer to f^1 as the *positive component* of f. The set f^X of inputs for which f is undefined can be computed as $f^X = \overline{f^1 + f^0}$. By definition, it is true that $f^1 \cdot f^0 = 0$. Boolean operations extend to TVFs in a trivial way. For example, for an AND gate with inputs a and b and output c, the TVF for c is given by $c^1 = a^1 \cdot b^1$ and $c^0 = a^0 + b^0$.

Malik computes the TVF for each node as follows. First, he finds a feedback arc set. By breaking these arcs and introducing auxiliary inputs y and outputs Y, the circuit becomes acyclic (see Figure 2). Next, ternary symbolic simulation is used to find the TVFs of the outputs z and Y. At the start of the simulation, the original inputs are initialized to boolean symbolic variables, and all other nodes (including y) are initialized to X (i.e., no assumption is made about the initial state of the circuit). The gates are evaluated in topological order from the inputs to the outputs. Evaluating a gate means to compute the output TVF of the gate by applying the function of the gate to its input TVFs. This is just simulating forward deduction in constructive logic.

After the first symbolic simulation pass of the acyclic circuit, the TVF at Y is fed back to y, and another pass is made. This continues until one pass completes without any TVF changing. It is sufficient just to monitor Y for change. Because of monotonicity over a finite lattice, convergence is guaranteed within k passes, where k is the size of the feedback arc set.

After convergence, the TVFs for z are examined: if there exists an input assignment that makes at least one output z_i evaluate to X, then the circuit is declared non-constructive; otherwise, it is declared constructive. If a circuit is declared constructive, then the function z_i^1 gives the boolean function for output z_i. Since z_i^1 is represented by a BDD, which can be trivially transformed into an acyclic circuit, the procedure can produce an acyclic version of the initial circuit. If a circuit is declared non-constructive, then $\sum_i z_i^X$ gives the input assignments for which at least one output does not constructively evaluate to a unique boolean value.

The correctness of Malik's procedure with respect to our notion of constructive circuits follows directly from the fact that it symbolically implements Scott's fixed point computation [2].

3.2 Our procedure

In our procedure, we use a method different from Malik's to compute the TVFs of the circuit nodes. Our goal is to minimize the number of gate evaluations performed during ternary symbolic simulation. As far as correctness

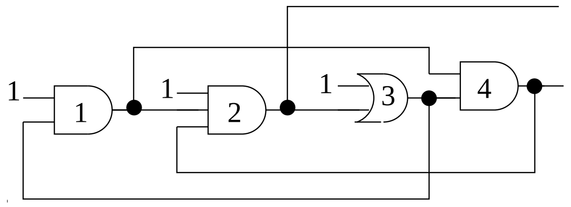

Figure 3: Number of gate evaluations depends on evaluation order.

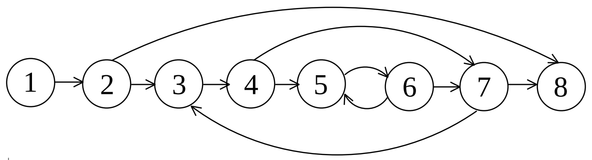

Figure 4: A directed graph.

is concerned, it turns out that the gates of the circuit can be evaluated in any order. The only requirement is that the process of evaluating gates continues until a fixed point is reached. On the other hand, the number of evaluations is sensitive to the order. Consider the circuit in Figure 3. Even though our algorithm performs symbolic simulation, suppose for a moment that we want to simulate the circuit for the concrete input 1,1,1. All internal nodes are initialized to X. Say we break the feedback at the outputs of gates 3 and 4, and then evaluate the gates in the order 1, 2, 3, 4. This would require 3 passes to reach convergence. However, if we break the circuit at the output of 2 and use the order 3, 1, 4, 2, we reach convergence in a single pass.

We apply an evaluation ordering scheme by Bourdoncle [3] to heuristically minimize the number of gate evaluations. Bourdoncle's algorithm takes as input a directed graph and produces a *weak topological ordering* (WTO). A WTO can be thought of as a decomposition of a graph into recursive, strongly connected components (SCCs). Consider the graph in Figure 4. The set $\{3, 4, 5, 6, 7\}$ forms an SCC. By removing vertex 3 from the subgraph induced by this set, we see that $\{5, 6\}$ forms an SCC. In this way, we can get nesting of SCCs. The WTO for the example graph is

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6)\ 7)\ 8.$$

The elements within a matching pair of parentheses constitute a *component*, and the first element of a component is the *head* (heads are underlined above). The *depth* of an element is the number of nested components containing the element (e.g. element 3 has depth 1; 6 has depth 2). The important properties of a WTO are that 1) each component is strongly connected, 2) the set of heads constitutes a feedback node set (that is, all backward edges are incident upon heads), and 3) it gives a total ordering on all the nodes.

Bourdoncle proposed a gate-evaluation order using a recursive strategy whereby an inner component is stabilized each time one pass is made of its containing component. So in the above example, we first evaluate 1, 2, 3, 4, 5, 6. But then, instead of going to 7, we return to 5, and continue looping between 5 and 6 until there is no change. Then 7 is evaluated, and then we return to 3. The process repeats until the component (3 4 (5 6) 7) is stabilized, and lastly

8 is evaluated. Bourdoncle showed that the total number of evaluations is bounded by $\sum depth(v)$, where the sum is taken over all nodes. This contrasts to the method that Malik uses, which is bounded by $N(k+1)$, where N is the number of gates and k is the number of feedback arcs. It can be shown that $\sum depth(v) < N(k+1)$. However, for a given circuit, both methods may converge faster than these bounds, and it is possible that Malik's method may converge sooner.

We compute the TVF for each node by evaluating the gates in recursive order. In addition to using Bourdoncle's method, we employ event-driven ternary simulation to further reduce the number of evaluations. With this technique, a gate is scheduled for evaluation only if the TVF of one of its fanins has changed. Once we have the TVF for each node, we examine the TVFs as explained above to determine whether a circuit is constructive.

4 Analysis of circuits with latches

Now we analyze circuits with latches. Such a circuit computes a sequence of output vectors from a sequence of input vectors. In the logical domain, a latch acts as an elementary logical delay. If a and b are the input and output of a latch, then $b_{n+1} = a_n$ for all $n \geq 0$. The initial output b_0 of a latch is assumed to be either 0 or 1, or nondeterministically 0 or 1, and is specified independently. In the electrical domain, a latch takes a clock as an additional input, and all latches are driven by the same clock. The input voltage is transferred to the output at each clock event. For a circuit with latches to work properly, there must exist a uniform stabilization delay that makes all outputs stabilize for any input vector, and the interval between clock events must be greater than this delay. Such a delay obviously exists for acyclic circuits, and can be shown to exist for constructive circuits [2].

We shall assume that combinational wires do not remember their value from one clock cycle to the next. This assumption is actually conservative for the usual electrical model, but it fits well with the semantics of synchronous languages such as Esterel or Lustre [8, 1], and with software implementations where intermediate wires are implemented by automatic variables rather than by static variables. Discarding this assumption is possible but leads to a more complex analysis since combinational wires can behave as "hidden state variables."

We say that a circuit with latches is *constructive* if the combinational part of the circuit is constructive when the primary inputs are restricted to care input values, and the latch outputs are restricted to reachable states. We have implemented a procedure to decide whether a circuit with latches is constructive. The procedure takes three inputs: 1) a circuit, 2) the set of initial values of the circuit latches, and 3) the care set of primary input values. If the circuit is constructive, the procedure returns "YES" and produces an equivalent acyclic circuit. Otherwise, it returns "NO" and produces an error trace showing how the circuit can be driven into a non-constructive state. Figure 5 shows the outline of the procedure.

For a circuit to be constructive, it is sufficient that the combinational logic block is acyclic. Thus, a depth-first search is first performed on the combinational logic; if no cycle is found, the circuit is constructive, and full constructivity analysis can be avoided. Otherwise, the following three step procedure is performed:

1. Find the TVF for each node in the circuit.
2. Compute the primary input/present state combinations which make the circuit non-constructive. This set of combinations is called the *unstableDomain*.
3. Determine whether any care input/reachable state combination is present in the *unstableDomain*. If not, the circuit is constructive.

In the following, we discuss the details of the three steps. Also, we discuss the mechanisms to generate acyclic implementations and error traces.

Step 1: Find TVFs for combinational circuit nodes. We can use either of the procedures from Section 3 to compute the TVFs. Latch outputs are treated as primary inputs, and hence are initialized to boolean symbolic variables. Since we make the assumption that wires do not remember their values from one clock cycle to the next, it is appropriate to initialize all combinational nodes to X, as is done in the procedures from Section 3.

This step effectively analyzes the behavior of the combinational part of the circuit for an arbitrary clock cycle, using boolean symbolic variables as inputs. Since the combinational wires do not hold state, there is no information that needs to be passed from one clock cycle to the next, except for the values of the latches.

Step 2: Compute the *unstableDomain*. The result of step 1 is the TVF for each node in the circuit. As a reminder, the TVF for a node gives the ternary value of that node for each boolean assignment to the primary inputs and present state signals. We are interested in the TVFs for the set T of primary outputs and next state signals. We define the *unstableDomain* as those boolean assignments to primary inputs and present state signals such that some signal in T evaluates to X:

$$unstableDomain = \sum_{t \in T} t^X .$$

Next, we intersect the *unstableDomain* with the *careSet* of primary input values, and then we project this set onto the present state signals to yield the set of *unstableStates*. This sequence of computations is illustrated in Figure 6. If *unstableStates* is empty for a given circuit, then we can immediately declare the circuit to be constructive. Otherwise, we must perform reachability to determine whether a member of *unstableStates* is reachable; this is checked in Step 3.

The definition of *unstableDomain* allows internally contradictory behavior, as long as it does not affect the combinational outputs. We can strengthen the definition to require *all* nodes in a circuit to evaluate to a known, boolean value. In this case, we assign T above to be the set of all nodes; we call this *strong constructivity*. The Esterel v4 compiler uses strong constructivity.

Step 3: Determine whether an unstable state is reachable. Steps 1 and 2 analyze the circuit for a single, but arbitrary, clock cycle. Step 3 effectively examines multiple clock cycles to determine whether a state in *unstableStates* is reachable via a sequence of care inputs. To do this, we use a standard symbolic traversal technique [6], limiting the

```
**function** constructivityAnalysis(circuit, initialStates, careSet)
  **if** (isAcyclic(circuit))
     **return** "YES"
  **else**
     **if** (isConstructive(circuit, initialStates, careSet, &unstableDomain, &reachabilityInfo)
        produceEquivalentAcyclicCircuit(circuit)
        **return** "YES"
     **else**
        getErrorTrace(circuit, unstableDomain, reachabilityInfo);
        **return** "NO"
```

Figure 5: Outline of constructivity analysis procedure.

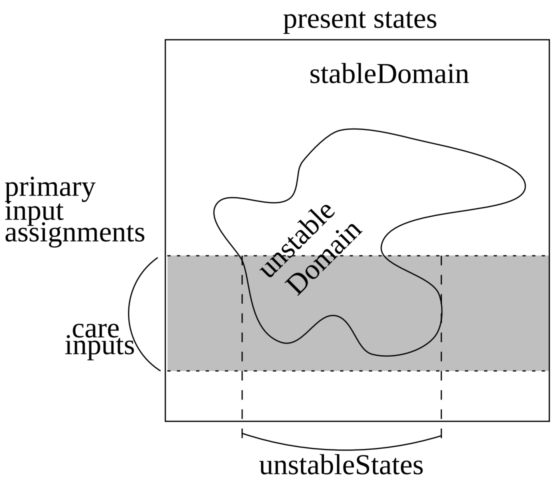

Figure 6: Computation of *unstableStates*.

next state computation to the input care set. However, there is one wrinkle: what do we use for the next state functions? The reachability computation requires a boolean function for each next state signal f, but Step 1 yields a ternary valued function. The solution is simple: we use the positive component f^1 of the TVF. As long as we stay within the *stableDomain*, f is guaranteed to evaluate to 0 or 1, and hence f^1 gives the correct value. Thus, upon reaching each new set of states during the reachability computation, we check that all these states are stable. If so, we are still within the *stableDomain*, and can use the positive component of each TVF to compute the next set of states. If not, we have found a reachable unstable state and can immediately conclude that the circuit is non-constructive. If we reach a fixed point without reaching an unstable state, then the circuit is constructive.

Generating an acyclic implementation. If the circuit is constructive, then we never leave the *stableDomain*. Hence, the positive component f^1 of each TVF gives the corresponding boolean function which needs to be implemented. Since we represent these boolean functions by BDDs, we can easily translate the BDDs to multi-level acyclic circuits. Roughly, we do this by interpreting a BDD vertex as a 2-to-1 multiplexer and grouping adjacent vertices into a single gate in the new circuit.

Generating an error trace. If the circuit is non-constructive, then the reachability step gives us a reachable unstable state. We simply work backwards, restricting ourselves to care inputs, to find a sequence of states starting from an initial state and leading to the unstable state. The process of generating an error trace is the same as that used in formal verification tools [10]. We also report the value of each circuit node on the "bad" assignment formed from the unstable state and a care input in the *unstableDomain*.

5 Application to Esterel

Esterel is an imperative synchronous language dedicated to reactive and real-time applications [8, 1]. The Esterel v4 compiler translates an Esterel program into a control circuit and a data path, which can both be implemented either in hardware or in software. We explain why Esterel programs can generate well-behaved cycles, and give experimental results.

5.1 Cycles in Esterel

The unit of communication in Esterel is the *signal*. At each cycle, a signal `S` is either absent (0) or present (1). By default, `S` is absent; it is made present by executing the statement "`emit S`". Signals are tested for presence by the statement "`present S then` p `else` q `end`" that combinationally transfers control to either p or q when it is executed. In the circuit implementation, a signal is implemented by an OR gate. A statement such as "`present S then emit T end`" builds a combinational path from `S` to `T`. Assume `I` is a primary input and consider the following statement:

```
present I then
   present S then emit T end
else
   present T then emit S end
end
```

There is a path from `S` to `T` and a path from `T` to `S`, hence a cycle. However, it is obvious from the source code that only one path can be used at a time, and, therefore, that the circuit is well-behaved.

Explicit delays also make cycles well-behaved. Consider for example:

```
present S then emit T end;
await I;
present T then emit S end
```

The three statements are in temporal sequence (operator "`;`"), and the "`await`" delay operator explicitly introduces a non-zero delay that makes the static `S`–`T` cycle dynamically harmless.

5.2 Experimental results

Our procedure for analysis of circuits with latches has been implemented in the Esterel v4 compiler, along with some improvements such as early garbage collection of TVFs. It has been applied to industrial-sized cyclic programs occurring in avionics applications at Dassault-Aviation. One program to which we applied our procedure has 1043 lines of actual Esterel code (comments excluded), 3989 wires, 157 latches, and 7 non-nested cycles. It is processed in 11.5 minutes of CPU time on a DEC AlphaServer 2000. A more pathological example, where the weak topological ordering has a depth of 12, exhausted the available memory using the procedure as described. However, by interleaving the constructivity and reachability analysis, this example also was successfully handled.

We hope to achieve better performance for big Esterel programs by adding additional optimizations. In particular, during the translation from programs into circuits, we can build a superset of the reachable states. This set can be used to simplify the intermediate TVFs by using the BDD *restrict* operator [6]. Preliminary results show that this may significantly reduce memory size.

6 Conclusion

We have described a procedure for determining if a circuit with latches is constructive. We say that a circuit is constructive if each output of the combinational part evaluates to a unique boolean value for every care input/reachable state combination. If the circuit is constructive, an equivalent loop-free implementation is produced, and if not, an error trace is produced. The procedure has been implemented in the Esterel v4 compiler, and has been successfully applied to industrial-sized examples.

Acknowledgments

Thanks to Horia Toma for improvements to the algorithm, and for providing the experimental results. Also, thanks to Jean Christophe Madre and Olivier Coudert for their help with the implementation. This work was conducted at Digital Equipment Corporation, Paris, France, with support from INRIA and Dassault-Aviation. In addition, the first author was supported by the Semiconductor Research Corporation under a fellowship and contract 95-DC-324A.

References

[1] G. Berry and G. Gonthier. The synchronous programming language Esterel: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[2] G. Berry and T. R. Shiple. Constructive boolean circuits. *To Appear*.

[3] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer-Verlag, 1993.

[4] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[5] J. R. Burch, D. Dill, E. Wolf, and G. D. Micheli. Modeling hierarchical combinational circuits. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 612–617, Nov. 1993.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[7] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishing, 1993.

[9] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, September 1995. Como, Italy.

[10] R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 41–58. Springer-Verlag, June 1993.

[11] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, July 1994.

[12] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.