Quick Conservative Causality Analysis

Ellen M. Sentovich
Cadence Berkeley Laboratories
1919 Addison Street Suite 303-304
Berkeley, CA 94704-1144

Abstract

The causality problem is that of determining if a combinational circuit with cycles has acceptable behavior: that for all inputs the outputs are well-defined and stable. While the problem manifests itself at the **circuit** *level, it usually originates at the* **system** *level. It may arise when a system is designed as a collection of modules: when composed, cycles are discovered in the ensemble. One must analyze these cycles to correct possible errors or to capture the correct behavior appropriately for further synthesis. Previously published algorithms use iterated ternary logic simulation. This is correct and robust, but expensive and in many cases overkill. In this work, a more efficient but conservative algorithm is proposed based on applying standard logic synthesis techniques of increasing power. We present initial results to demonstrate the practicality of this approach.*

1. Introduction

1.1. The causality problem

The *causality* problem here is that of determining if a combinational circuit with cycles has *acceptable* behavior. Acceptable circuit behavior is often defined as behavior that has an input/output-equivalent acyclic circuit implementation. That is, for each input of the cyclic combinational circuit, the outputs are well-defined and stable. Any algorithm for causality analysis relies on a particular definition of acceptable behavior, which specifies equivalence and stability conditions. This behavior in turn relies on an underlying delay model. At a higher level, causality can be viewed as the property that a module has well-defined, stable behavior at the outputs for each possible input.

The causality problem may arise in several contexts:

- An *implicit* specification of a single circuit or state machine may not have a well-defined causal implementation (i.e., with an acyclic combinational part).
- A set of causal circuits composed together may exhibit non-causal behavior.
- In system specification, where the design is composed of individual modules, combinational loops between modules may be accidentally or deliberately (e.g., shared resources) designed in.

In many cases, the design has been well-conceived, and the cycles are in fact false: they are static cycles that are never active dynamically.

With ESTEREL[2] specifications, causality problems can arise in two ways. First, due to the powerful language constructs and implicit nature of state-machine specification, an individual module may exhibit causality problems. Second, during module composition, individually causal modules may produce a system with causality problems. A number of examples of the first type, some causal and some non-causal, are given in [1]. An example of the second type, which is a real bus arbiter that is causal but cyclic, is given in [4]. The specification has a circuit implementation that is statically cyclic but dynamically acyclic. Causality analysis is used in the compiler now to determine which systems are truly causal despite their structural cycles.

Another application is in behavioral specifications, where resources may be shared in such a way as to create a static cycle in the specification. An example is given in [6]: two operators ADD and SHIFT may be performed in either order depending on the value of a select variable. Hence there are paths from ADD to SHIFT and from SHIFT to ADD, but they are never active simultaneously.

When combinational cycles are detected, causality analysis is usually carried out before synthesis and optimization proceeds. If the circuit is correct (causal), it can be re-implemented as an acyclic circuit. If incorrect, one must find the undesirable behavior and correct it. One can certainly generate cyclic hardware and software implementations for cyclic specifications immediately, skipping causality analysis. However, without this analysis, there is no guarantee of correct behavior, and without generation of the equivalent acyclic implementation, there are currently no methods for

synthesis and optimization.

1.2. Previous work

In [6], a definition of combinational circuits[1] was given with an algorithm for determining if a cyclic circuit is in fact combinational. The algorithm is based on symbolic simulation using the 3-valued Scott Boolean domain: $\{0, 1, \perp\}$. It consists of breaking all cycles in the circuit, assigning broken wires the undefined value $\perp$, and iteratively simulating, assigning newly-determined simulation values to the broken feedback wires at each iteration. The algorithm is monotonic and guaranteed to converge. This paper also provides good motivation and references for the causality problem (e.g., [11] which solves the problem of cycles in the context of resource sharing by restricting the sharings considered).

In [8], the algorithm was extended to sequential circuits, and an efficient implementation was outlined. In [9], a proof of correctness of the algorithm under the up-bounded inertial delay model was given.

In parallel, the notion of *constructive causality* [1] was being developed and imposed as a requirement for correct ESTEREL specifications. In the ESTEREL language, one specifies a synchronous, reactive, control system as a set of interacting modules. The language is based on communication of signals, and provides constructs for concurrency and pre-emption. The ESTEREL compiler translates the set of modules into an equivalent Boolean circuit, which is another implicit representation of the underlying state machine.

The implicit specification method and rich signal constructs imply that non-causal behavior can be given. The compiler performs a static causality check on the input program translated to a dependency graph. This approximation to the definition of causality held at the time turned out to be too weak. As a result, constructive causality was born. Its complete semantics at the behavioral, operational, and circuit levels, is given in [1]. The basic idea is that a design must be *reactive* (have at least one response per input), *deterministic* (at most one response), and *constructive* (the solution is derived by fact propagation rather than self-justification). The key result is that *a constructively causal design has a stable, well-defined circuit equivalent*. Therefore, the complete causality analysis program in ESTEREL, `sccausal` [5], is based on the results in [6, 8, 9, 1] and is applied to the Boolean circuit representation of ESTEREL programs.

[1] A *combinational circuit* hereafter is a circuit with no delay elements that has "acceptable" behavior, as described in the beginning of this introduction and formally defined in [9].

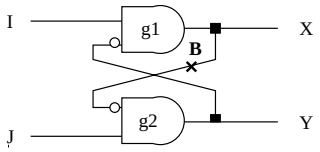


Figure 1. Simple non-causal circuit

1.3. Efficient causality analysis

We present here an algorithm that is built on the same assumptions (delay model, ternary simulation) but that uses standard logic synthesis and optimization. It is more efficient since it does not require a three-valued model, it does not perform iteration, and it uses a variety of efficient algorithms applied successively to the circuit rather than full symbolic simulation at the outset. It is also conservative in that it may declare a causal circuit non-causal. We focus on the analysis of cyclic combinational circuits to determine an acyclic equivalent. The extension to sequential circuits is not trivial, but has been done in [8]; that method can be easily modified to work with the causality analysis method proposed here. Of course, combinational causality analysis can be applied to sequential circuits directly, but the analysis will only be carried out statically: as though all states are reachable. We present results on real combinational and sequential circuits generated from ESTEREL.

2. Example

A simple example illustrates the notion of causality, causality analysis, previous algorithms, and the contribution of this work.

Consider the circuit shown in Figure 1. It is electrically non-causal: when I and J are both 1, $X = \overline{Y}$ and $Y = \overline{X}$. While stable solutions exist ($X = 1$, $Y = 0$; $X = 0$, $Y = 1$), there is no guarantee that the wires will stabilize given the delays, and there is no way of knowing which of the two stable solutions will be reached. The ternary simulation algorithm would break the cycle at, for example, point B, creating a new input and output, B_{in} and B_{out}. Assigning B_{in} to $\perp$ (unknown), and allowing the inputs to assume all values, we obtain $B_{out} = I \cdot \overline{Y} = I \cdot (\overline{J} + B_{in}) = I\overline{J} + I\perp$. Iterating, $B_{out} = I \cdot (\overline{J} + (I \cdot \overline{J} + I \cdot \perp)) = I\overline{J} + I\perp$. There is no change, so the iteration stops and the circuit is non-causal since B_{out} cannot be assigned a stable value for each value of the input.

Now consider a modification of this circuit, where the $\{1, 1\}$ input to the cross-coupled gates is prohibited by the environment. This is modeled by the circuit in Figure 2. Again breaking the cycle at B, $B_{out} = I'\overline{Y} = I' \cdot (\overline{J'} + B_{in}) = I\overline{J} \cdot (I + \overline{J} + B_{in}) = I\overline{J} + I\overline{J}\perp = I\overline{J}$ and stability is

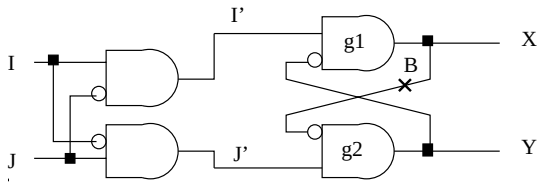


Figure 2. Restricted inputs creates a causal version

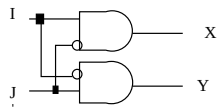


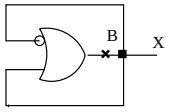
Figure 3. Acyclic version

reached. B_{in} is now implemented by $I\bar{J}$. The final acyclic version of the circuit is shown in Figure 3.

Now suppose we take the original circuits, cut the B arcs, and ask the question "is B_{out} independent of B_{in}?" For the non-causal circuit, $B_{out} = I\bar{J} + IB_{in}$, $B_{out_{B_{in}}} \overline{\oplus} B_{out_{\overline{B_{in}}}} = \bar{I} + \bar{J} \neq 1$ and the answer is no. For the causal circuit, $B_{out} = I\bar{J}$ and the answer is yes. This simple analysis is equivalent to a single iteration of the ternary symbolic simulation, but note that there are many ways of determining the independence of B_{out} from B_{in} using logic synthesis techniques. For example, in the simplest case, constant propagation in the circuit may reveal independence. Efficient ATPG techniques can be used to determine independence, as can logic simplification. In the limit, the BDD for B_{out} can be built and tested for independence: $B_{out_{B_{in}}} \equiv B_{out_{\overline{B_{in}}}}$? Or even more simply, does B_{in} appear as a variable in $BDD(B_{out})$? The independence check is of course computationally equivalent to exhaustive simulation of an output with respect to an input, but by viewing the problem this way, and noting that in many cases full analysis is not required, a better algorithm can be devised. Simple logic synthesis algorithms are not performed a priori in the ternary simulation analysis method because an alteration of the logic may destroy the causality properties of the circuit.

Such simple arc-breaking followed by dependency analysis is not by itself a correct algorithm for causality determination, as the next example illustrates.

Example 2.1 *Consider the following simple non-causal circuit.*



This circuit is reactive and deterministic, and hence logically correct. However, it is not constructively causal since it stabilizes only due to feedback. See [1] for the precise definition of constructive causality and the reasons behind this choice.

Breaking the two cycles by cutting before the fanout point results in $B_{out} = B_{in} + \overline{B_{in}} = 1$, which is not the correct result. Breaking the two cycles using the two feedback arcs does retain the dependency needed to demonstrate that this circuit is non-causal. The difference, of course, is precisely what one obtains using ternary versus binary simulation.

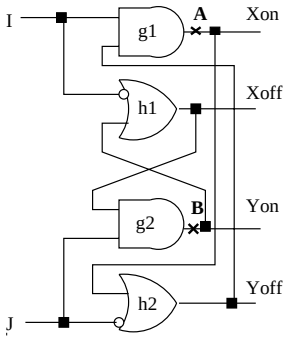


Figure 4. Dual-rail encoding of IJ-example

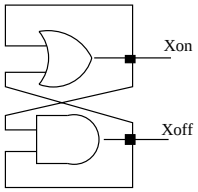
3. Method

Several observations about causality analysis and ESTEREL-generated circuits lead to our new algorithm, as described in this section.

Observation 3.1 *Ternary simulation is equivalent to dual-rail encoding and binary simulation.*

This is formally proven in [3]. This observation implies that in the complete causality analysis algorithm, one can replace a ternary simulation iteration by a transformation of the circuit to the dual-rail encoded version followed by a binary simulation. The dual-rail encoded version of the IJ-example circuit is shown in Figure 4. Note that each internal signal is encoded by a pair of signals, and that all the gates in the circuit are positive unate (no negations) with respect to the internal signals. In practice, it is most efficient to first break the cycles and dual-rail encode only the signals corresponding to the broken cycles, and all the signals in their transitive fanout cones.

Example 3.1 *The dual-rail encoding of the OR-gate with feedback is given by*



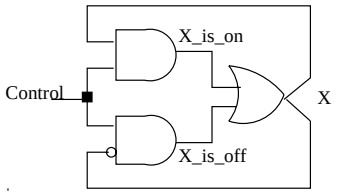


Figure 5. ESTEREL version of OR-gate example

The reader can verify that cycle breaking and dependency analysis will now yield the correct result: Xon_{out} *and* $Xoff_{out}$ *will depend on* Xon_{in} *and* $Xoff_{in}$ *respectively, regardless of where the feedback is broken.*

Observation 3.2 *Often in practice iteration is not necessary, especially if one breaks the feedback judiciously.*

This observation indicates that a non-iterating, conservative computation will often suffice to correctly compute causality. In [6], it was demonstrated that convergence is reached in $\leq k$ iterations, where k is the number of feedback arcs broken. It follows that if only one arc is broken, the non-iterating algorithm produces the correct result.

Thus, it is very important to choose the feedback set carefully. If simulation is iterated, a poor choice still leads to a correct result, but after many iterations. (For example, consider the case of breaking all the internal wires of a circuit.) If simulation is not iterated, fewer feedback arcs provides increased reliability of results as less information flows across the feedback boundary.

Observation 3.3 ESTEREL*-generated circuits are partially dual-rail encoded.*

The OR-gate example treated in the previous section might be specified in ESTEREL as follows:

```
module OR:

signal X in
present X then emit X else emit X end
end signal

end module
```

The direct translation to a circuit as described in [1] leads to the circuit shown in Figure 5.[2] While Xon and Xoff are not explicitly modeled by wires as they are for full dual-rail, the positive and negative tests for X being ON and OFF are explicitly modeled by wires X_is_on and X_is_off as shown in the figure. The present test of a signal in ESTEREL *always* generates two gates in this manner and hence two fanout arcs. Note also that fanout points are not modeled: gates have multiple fanout arcs. If the signal is on a cycle and its fanout does not reconverge, effectively both arcs must be broken to break all feedback (which is equivalent to dual-rail). If there is reconvergence, it is possible that only one arc is broken (where two must be broken in the dual-rail version). This may still return a correct causality result depending on the function at reconvergence, *but not always*. The circuits in Figures 1 and 2, for example, are correctly analyzed with one broken feedback arc. Still, we can take advantage of this pseudo-dual-rail property of ESTEREL circuits to build and test a quick prototype causality checker. In summary:

- Observation 3.1 implies that an iteration of the ternary simulation-based causality checking algorithm can be performed by first dual-rail encoding the circuit and then performing binary simulation. Furthermore, any binary-based transformations (e.g., standard logic optimization) can be performed on the dual-rail version without destroying causality properties.

- Observation 3.2 implies that in many cases, and certainly if only one arc is broken, a single simulation iteration suffices to determine causality. If multiple iterations are required for a causal circuit, at the completion of the first iteration the information is simply inconclusive: it will appear as though the values of the feedback arcs are unknown, when in fact further iteration would determine these values. Hence, single-iteration simulation is usually correct by observation 3.2, and at least conservative (a non-causal conclusion may be made for a causal circuit).

- Observation 3.3 implies that in many cases, one need not fully dual-rail encode a circuit to use the binary simulation method for causality determination: the ESTEREL-generated circuit already duplicates enough signals that arc-breaking and binary simulation suffices to determine causality. The result in this case is not conservative as a non-causal circuit may be deemed causal. However, one can use this notion to implement a prototype causality checker to compare the performance of a binary-simulation-based algorithm to that of the full ternary one.

These are the main ideas behind our method. The algorithm is described more thoroughly in the next section.

4. Algorithm

Observations from the previous section indicate that a conservative alternative algorithm to iterated ternary simulation would consist of:

[2] This is only part of the circuit generated. There is additional circuitry implementing initialization and the termination codes.

- breaking a minimal number of feedback wires
- transforming the circuit into the dual-rail encoded version by dual-rail-duplicating gates (adding the DeMorgan-equivalent gate) in the transitive fanout of each broken-arc input
- performing dependency analysis (binary simulation).

Observation 3.3 only indicates that one may omit the dual-rail encoding step and still obtain the correct result in some cases. (In all of our cases, we obtain the correct result, both for causal and non-causal circuits.) We use this observation to quickly build a prototype of our algorithm to test it against the current version, though in practice this is of less use since it is not conservative, and thus conservative or full causality will have to be run subsequently anyway (except in the cases where it determines a circuit to be non-causal, and produces a meaningful error trace). The prototype algorithm simply

1. breaks cycles
2. performs a dependency computation
3. creates an acyclic equivalent version if possible, or produces an input pattern exhibiting the causality error.

4.1. Breaking combinational cycles

As noted in Section 3, it is desirable to break as few arcs as possible. We implemented two algorithms for cycle-breaking. The first is guaranteed to break the minimum number of arcs, and was published in [10]. It is very fast even on large graphs. The second is the very simple but fast depth-first search algorithm of Tarjan. In our experiments, as we had quite small examples or large examples with small strongly connected components, we always used the more robust exact algorithm.

4.2. Dependency computation

The second step in the algorithm is to determine whether or not the broken feedback outputs logically depend on the broken feedback inputs; if not, they can be expressed independently, and the broken arcs reconnected to create an acyclic equivalent circuit.

The dependency computation interleaves logic optimization and dependency check. The simple dependency check processes nodes in topological order searching for a path from each B_{in} to each B_{out} and using this information to compute on-the-fly whether cycles would appear in the circuit were the arcs to be reconnected. If not, the check is successful. If only a single arc is broken, this computation simplifies to a check for a path from B_{in} to B_{out}. The complete dependency computation proceeds as follows:

- simple constant propagation; simple dependency check
- simple logic optimization, collapsing small nodes together and simplifying node functions; simple dependency check
- introduce external don't care conditions from the ESTEREL specification and repeat simple logic optimization; simple dependency check
- build BDDs for all B_{out} variables; check for BDD dependencies of each B_{out} on its corresponding B_{in}, determine the existence of cycles.

The last is the most robust, and returns an exact answer as to whether an output depends on an input. It is not too expensive in practice, as the strongly connected components in the original circuit graph are small (and hence the logic cones between temporary inputs and outputs), and the circuit has already been optimized before building the BDDs.

In general, it is not sufficient to check simply that each B_{out} is independent of its corresponding B_{in}. With more than one feedback arc, interdependencies could still lead to a cyclic circuit on reconnection. For this, it suffices to build a dependency graph using the dependency information computed above (from simple to robust) and perform acyclicity checks on this graph. The on-the-fly cycle check mentioned above is an efficient implementation of this. In practice, we never observed interdependencies leading to additional cycles.

4.3. Creating the acyclic circuit or producing an error trace

If the simple dependency check is successful, an acyclic version of the circuit can be created by simply reconnecting the broken feedback arcs. A negative simple dependency check result implies there is no path from the input to the output of interest (or no cycle-forming set of paths), so simple reconnection will result in an acyclic circuit. (This will only happen after some logic alteration, such as constant propagation. Before this, it is known that there is at least one topological path between the two or there would not have been a cycle.)

If simple dependency fails but BDD dependency succeeds, the acyclic circuit must be created using logic functions derived from the BDD and then reconnecting the feedback arcs. That is, the function for each feedback arc output is completely re-implemented based on its BDD. This step can imply a significant logic increase depending on the functions: the ADD/SHIFT example mentioned in the introduction would require duplication of these operators. In practice, this step is not too expensive as the BDDs are limited to only the necessary scope in the circuit. Furthermore,

our examples were limited to circuits described in ESTEREL, which tend to be control-based.

If the circuit is dual-rail encoded before the dependency check, a circuit equivalent to the original is obtained by merging the dual rail (feedback) inputs and removing the negative dual rail outputs before reconnecting the feedback arcs. This is followed by simple logic optimization and/or explicit merging of the duplicated internal signals to recover the area overhead incurred in dual-rail encoding.

If the circuit is determined to be non-causal, an error condition is produced simply by: *Input_error* = $B_{out}{}_{B_{in}} \oplus B_{out}{}_{\overline{B_{in}}}$. For sequential circuits, an additional constraint is imposed that the *Input_error* contain the initial state, to ensure that it is a valid condition.[3]

4.4. Optimizing cyclic circuits

We have cast the algorithm as one for causality analysis, but it can be used as a cyclic circuit optimizer as well. Suppose one has a cyclic implementation, and one would like to retain this form since it is more efficient. There are currently no logic optimization programs that can optimize cyclic circuits. In addition, for the same reason that ternary simulation must be used to perform causality analysis, acyclic subcircuits of a cyclic circuit cannot be abstracted and optimized with standard techniques while maintaining the causality properties. The simple algorithm proposed here is a cyclic circuit optimizer: cycles are dual-rail encoded, standard logic optimization is applied, dual-rail signals and gates are merged to recuperate the overhead, and a final logic optimization is applied.

5. Implementation and results

The program `cheap_cause` has been implemented inside the SIS logic synthesis program [7]. A dependency graph is created for the cyclic circuit and used to compute the set of feedback arcs. The corresponding wires are broken in the network and the dependency analysis performed. Dual-rail encoding was not implemented for this prototype implementation; in fact, with the simple reliance on the partial dual-rail encoding obtained with the ESTEREL translation, the correct causality result was obtained in all cases. Furthermore, the advantage of this method in terms of final logic area will not be affected by the overhead incurred by dual-rail encoding: duplicated gates and signals can later be merged.

[3] It may be valid in another state, but we do not yet do the analysis to determine the valid reachable states. In all experiments, an error condition was found at the initial state.

5.1. Comparison with `sccausal`

For comparison, we describe the `sccausal` algorithm [5] and note some of its properties. First, three-valued functions (TVFs) for each node computed using two variables per node, and using a BDD representation. This is expensive since the number of variables is doubled; it has a dramatic effect on the efficiency of the BDD computations. Next, a weak topological ordering (WTO) is computed for the nodes: rather than compute feedback arcs directly, an ordering is computed for node processing during simulation. This ordering is not a static one-pass through all nodes, but rather contains cycles within that are iterated to convergence. The algorithm iterates to convergence, correctly handles full ESTEREL (the algorithm described here handles only pure ESTEREL), and handles sequential circuits. Finally, if the circuit is causal, an acyclic version is built based on the BDDs. This is another source of inefficiency, since the structure of the initial implementation is lost completely.

5.2. Experiments

A number of experiments have been run on the prototype version of `cheap_cause`, and comparisons made with `sccausal`. However, we note the following points. First, neither `sccausal` nor `cheap_cause` are fully optimized: `sccausal` has been released in a beta version, and `cheap_cause` is still a prototype. Second, though in some cases `cheap_cause` may return the correct result immediately (in all cases, in our examples), even with full dual-rail encoding it is conservative since it performs no iteration. Therefore, its utility is intended more as a preprocessor to `sccausal`, and as an optimization scheme for cyclic circuits, than as a causality analysis program in its own right.

We have run our program on all the anomalous causality programs `P1`–`P13` described in [1]. In all cases, `cheap_cause` returned the correct result very quickly. Comparison of computation times with `sccausal` is not meaningful since the circuits are so small.

The program was applied to an industrial example for in-vehicle communication [12] called **prosa**. The original version is causal but has cycles. `sccausal` makes this determination and computes the acyclic circuit in 720 seconds on a sun4. `cheap_cause` was able to correctly determine this with only one broken arc (the original graph of 1989 vertices and 3463 edges is reduced to a single strongly connected component with 82 vertices and 118 edges), and simple constant propagation, in 19.4 seconds. The circuit produced by `sccausal` has 8346(6842) literals in sum-of-products (factored) form, and 64 latches; after optimization in SIS, it has 1099(839) literals and 44 latches. The circuit produced by `cheap_cause` has 1268(1268) literals and 73

latches; after optimization, 838(761) literals and 71 latches.

On a second, non-causal version of **prosa**, `sccausal` completes in 987 seconds, `cheap_cause` in 61 seconds. The error traces cannot be compared since `sccausal` returns values on internal **sc** format wires and corresponding lines in ESTEREL code (useful for source-code debugging), and `cheap_cause` returns values on the circuit I/O signals (useful in the ESTEREL simulators).

The second example, **mejia**, is an industrial control design which is causal but contains cycles. `sccausal` performs the analysis and writes the result in 44 seconds. The circuit has 2910(2454) literals and 28 latches, reduced to 637(484) literals and 21 latches by SIS. `cheap_cause` breaks one arc and performs the analysis in 25 seconds, with a resulting circuit of 878(878) literals and 43 latches, reduced to 543(439) literals and 43 latches by SIS.

It is interesting to note the difference in the number of registers. `sccausal` is able to reduce this number on the fly during the reachable states computation. In both cases, the initial circuit is considerably smaller for `cheap_cause`, which could make a decidable difference for further optimization of large circuits.

These are of course good scenarios on the type of designs that `cheap_cause` can handle. We mention these results simply as indication that the proposed algorithms have practical use; more experimentation is warranted.

6. Conclusions and future work

We have presented an algorithm for efficient causality checking. While it is conservative, it has been demonstrated on a suite of examples to return correct results. It is based on the observation that ternary simulation is equivalent to dual-rail encoding of feedback signals followed by binary simulation. The latter is implemented efficiently by applying a series of logic optimization techniques of increasing power. The results provide both a method for quick causality analysis, and a method for optimization of cyclic circuits. Furthermore, the final acyclic implementation may be produced directly from the logic synthesis tools rather than from BDDs, so it retains the structure of the initial implementation as much as possible.

There are several areas for future work. First, full cheap causality needs to be implemented with complete dual-rail encoding of the circuits, rather than relying on the structure of ESTEREL-generated circuits and the arc-breaking algorithm to maintain some integrity of the causality analysis. Second, a modified algorithm must be developed for sequential circuits. This algorithm will simply be an iteration of the causality analysis presented here, and a reachable states computation (both done on the arc-broken acyclic circuit) similar to the algorithm published in [8]. Third, more examples for causality analysis must be obtained and analyzed, so that the gain of this new method over the complete method can be assessed on practical designs. Finally, the techniques should be applied to cyclic circuits as an a priori optimization method, before full-causality is carried out. This should improve the performance of `sccausal` significantly.

Acknowledgements

Helpful discussions with Gérard Berry are gratefully acknowledged. This work was supported in part by the National Science Foundation under grant INT-9505943.

References

[1] G. Berry. *The Constructive Semantics of Pure Esterel*. 1996. To Appear, available now at ftp://www.inria.fr/meije/esterel/papers/constructiveness.ps.gz.

[2] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] G. Berry and E. Sentovich. On constructive causality, 1997. Work in progress.

[4] R. DeSimone. Note: A small hardware bus arbiter specification leading naturally to correct cyclic description, Mar. 1996. Technical note.

[5] A. Girault, T. Shiple, and H. Toma. The sc-causal compiler, 1997. Documentation provided with the ESTEREL compiler.

[6] S. Malik. Analysis of Cyclic Combinational Circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 618–625, Nov. 1993.

[7] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc of the ICCD*, pages 328–333, Oct. 1992.

[8] T. Shiple, G. Berry, and H. Touati. Constructive Analysis of Cyclic Circuits. In *Proceedings of the European Design & Test Conference*, pages 328–333, Mar. 1996.

[9] T. R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, UC Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, Oct. 1996. Memorandum No. UCB/ERL M96/76.

[10] G. Smith and R. Walford. The Identification of a Minimal Feedback Vertex Set of a Directed Graph. *IEEE Transactions on Circuits and Systems*, CAS-22(1):9–15, Jan. 1975.

[11] L. Stok. False Loops through Resource Sharing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 345–348, Nov. 1992.

[12] R. v. Hanxleden, J. Bohne, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software co-design of a fault-tolerant communication protocol. In *Proceedings of the IEEE International Workshop on Embedded Fault-Tolerant Systems*, Dallas, Sept. 1996.