Symbolic Model Checking for Sequential Circuit Verification

J. R. Burch E. M. Clarke D. E. Long
K. L. McMillan D. L. Dill

July 15, 1993
CMU-CS-93-211

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, and in part by the U.S. Air Force under Contract F33615-90-C-1465, ARPA Order No. 7597, and in part by the National Science Foundation under contract numbers CCR-8722633 and MIP-8858807, and in part by the Semiconductor Research Corporation under Contract 92-DJ-294. The fourth author was supported by an AT&T Bell Laboratories Ph.D. Scholarship. The fifth author was supported at Stanford University by a CIS Seed Research Grant.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, the SRC, or the U.S. government.

Contents

1 Introduction **4**
- 1.1 Contributions 4
- 1.2 Related Work 5

2 Binary Decision Diagrams **7**

3 Representing Circuits **8**
- 3.1 Synchronous Circuits 9
- 3.2 Asynchronous Circuits 10
- 3.3 Partitioned Transition Relations 11

4 Finding Reachable States **12**
- 4.1 Frontier Set Simplification 13
- 4.2 Iterative Squaring 14
- 4.3 Invariants 15

5 Computing Relational Products **17**
- 5.1 Basic Algorithm 17
- 5.2 Disjunctive Partitioning 19
- 5.3 Conjunctive Partitioning 19
- 5.4 Recombining Partitions 21

6 Symbolic Model Checking **22**
- 6.1 Computation Tree Logic 22
- 6.2 Model Checking 23
- 6.3 Fairness Constraints 24

7 Verifying Synchronous Circuits **26**
- 7.1 Pipelined ALU 26
- 7.2 Other Synchronous Examples 33

8 Verifying Asynchronous Circuits **35**
- 8.1 Modified Breadth First Search 35
- 8.2 An Asynchronous Stack 37
- 8.3 Distributed Mutual Exclusion 40

9 Discussion **43**
- 9.1 Transition Relations 44
- 9.2 Degree of Automation 45

Abstract

The temporal logic model checking algorithm of Clarke, Emerson, and Sistla [17] is modified to represent state graphs using *binary decision diagrams* (BDDs) [7] and *partitioned transition relations* [10, 11]. Because this representation captures some of the regularity in the state space of circuits with data path logic, we are able to verify circuits with an extremely large number of states. We demonstrate this new technique on a synchronous pipelined design with approximately 5×10^{120} states. Our model checking algorithm handles full CTL with fairness constraints. Consequently, we are able to express a number of important liveness and fairness properties, which would otherwise not be expressible in CTL. We give empirical results on the performance of the algorithm applied to both synchronous and asynchronous circuits with data path logic.

1 Introduction

Bugs found late in the design phase of a digital circuit are a major cause of unexpected delays in realizing the circuit in hardware. As a result, interest in formal verification techniques for hardware designs has been growing. A number of different methods have been proposed, but nearly all can be classified in terms of the natural division between the *data paths* and the *controlling circuitry* in digital circuits. The most successful methods to date for verifying data path logic treat only functional behavior, without considering sequential behavior. These methods are frequently based on the use of automatic theorem provers or proof checkers and may require considerable assistance from the user in constructing a correctness proof. In contrast, the most effective techniques for reasoning about sequential behavior usually require a complete exploration of the state space of the circuit [6, 21, 25]. The state exploration techniques are attractive because they are highly automatic: the user simply provides a description of the circuit implementation and its specification; the system does the rest. In the case of a single controller, the approach is often quite practical, since the number of states tends not to be excessive. The approach has not been very useful with data paths, however, since the number of states is almost always too large to permit explicit enumeration. In order to reason about the complex interaction between controllers and data paths we need techniques that are able to handle both types of circuits. Developing such techniques has proven to be a very difficult problem. However, the regularity of data path designs provides some reason to believe that their state graphs, while large, will often have a relatively simple structure. Consequently, it may be possible to find a concise representation that exploits the uniformity of the state space and depends in size more on the inherent complexity of the data path logic than simply the number of states it determines.

In this paper, we show how *temporal logic model checking* [12, 13, 14, 16, 17] and *reachability analysis* algorithms can modified to represent state graphs using *binary decision diagrams* (BDDs) [7]. Because this representation captures some of the regularity in the state space determined by sequential circuits, we are able to verify sequential circuits with an extremely large number of states. The algorithms are based on computing fixed points of functions, called predicate transformers, that map sets of states to sets of states. The predicate transformers are used to describe properties of circuits and are derived from the transition relations of the circuits. Both state sets and predicate transformers are represented with BDDs. Thus, we are able to avoid explicitly constructing the state graph of the circuit. We have tested the performance of the algorithms on both synchronous and asynchronous circuits with data path logic. We were able to verify a pipelined ALU with over 10^{120} states and an asynchronous stack with over 10^{50} states. More importantly, for the classes of circuits that we verified, the CPU time required increased as a small polynomial in the number of components of the circuit. These results provide strong evidence of the scalability of our methods.

1.1 Contributions

The major contributions of this paper are as follows.

1. A BDD-based algorithm for CTL model checking with *fairness constraints.*

2. A description of *disjunctive partitioned transition relations* and *conjunctive partitioned transition relations.* With these methods, computing the transition relation of a circuit never limits the size of the circuits that can be verified.

3. A *modified breadth first search* algorithm to speed up reachability analysis for circuits represented with disjunctive partitioned transition relations.

4. A thorough empirical study of the asymptotic complexity of our methods using several substantial examples.

5. General techniques for improving the efficiency of verification methods based on reachability analysis by viewing such verification as automatically constructing and checking an invariant.

Several of the above contributions are full length descriptions of results the current authors first described in the conference literature [10, 11, 12, 13, 14].

1.2 Related Work

There are a number of approaches for verifying sequential circuits by state exploration techniques. Not long after Bryant described BDDs [7], several groups began adapting state exploration algorithms for use with BDDs.

Coudert, Berthet, and Madre developed a method for showing equivalence between two deterministic finite automata [18]. Given two automata, they perform a breadth first search of the state space of the product automata. BDDs are used to represent sets of states and the possible transitions of the automata. For the latter, a *transition function vector* is used. This is a vector of BDDs, one for each state bit, that represents the next state logic of the circuit. Cho *et al.* [15] discuss a similar technique.

Several groups have independently applied BDDs to CTL model checking [16, 17]. Burch, Clarke, McMillan and Dill [12] have developed a symbolic CTL model checker that uses transition relations to represent the circuit being verified. Coudert *et al.* [20] and Bose and Fisher [3] have described BDD-based algorithms for CTL model checking that use transition function vectors for this purpose. Since all three of these verification techniques are based on CTL, they are able to handle specifications that include unbounded liveness properties. Such specifications cannot be handled by other symbolic techniques for sequential circuit verification such as those described by Bryant and Seger [9], Bose and Fisher [2], and Coudert *et al.* [18]. In addition, the algorithm of Burch *et al.* permits arbitrary CTL formulas to be used as *fairness constraints* [17].

A serious limitation of the approaches that use transition function vectors, as opposed to transition relations, is that they cannot model nondeterministic systems in a natural way. When modeling systems for verification, there are two major sources of nondeterminism. First, nondeterminism can occur because of concurrency in the underlying circuit (as is the case in most asynchronous circuit models). Second, nondeterminism arises when abstraction is used to simplify reasoning about some part of the circuit. Because abstraction may hide part of the state of circuit, a transition may appear nondeterministic even though it

was originally deterministic. As an example of these two situations, consider the cache coherency protocol for the Encore Gigamax that McMillan has investigated [29, 30, 31]. The protocol was designed for a shared memory multiprocessor organized as a series of buses connected by an asynchronous hierarchical routing network. The caches on each bus are kept consistent using bus snooping, while a complex message passing protocol is used to ensure consistency between caches on different buses. McMillan modeled the system as an asynchronous composition of synchronous finite state machines. He also used abstraction to simplify the verification, which made the model even more nondeterministic. For example, McMillan did not precisely model the cache replacement mechanism. When modeling asynchronous systems or using abstraction, it is often necessary to use fairness constraints to make accurate models. For example, fairness constraints are required to describe a gate with an arbitrary but finite delay.

If a single BDD is used to represent a transition relation, the size of the BDD can become a bottleneck. This problem can be solved for asynchronous circuits by representing the transition relation as an implicit disjunction of BDDs [12], a technique we now call *disjunctive partitioned transition relations*. Adapting this technique to synchronous circuits requires *conjunctive partitioned transition relations*. Touati *et al.* [34] and Burch, Clarke and Long [10, 11] developed methods for computing an image of a conjunctive partitioned transition relation (the latter method is described in section 5). The efficiency of both techniques derives from *early quantification* of state variables. We believe that our technique often allows more early quantification, and so is more efficient. The available empirical results support this conclusion, although more experimentation is necessary before a definitive conclusion can be reached. A detailed comparison of the two methods is presented in section 9.

Bryant, Seger and Beatty [8, 9] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic. A specification consists of preconditions and postconditions expressed in the logic. The preconditions are used to restrict inputs and initial states of the circuit; the postcondition gives the property that the user wishes to check. Formulas in the logic have the form

$$p_0 \wedge \mathbf{X}p_1 \wedge \mathbf{X}^2 p_2 \wedge \cdots \wedge \mathbf{X}^{n-1} p_{n-1} \wedge \mathbf{X}^n p_n.$$

Note that the syntax of the formulas is highly restricted compared to most other temporal logics used for specifying programs and circuits. In particular, the only logical operator that is allowed is conjunction, and the only temporal operator is *next time* ($\mathbf{X}$). However, the logic is still applicable to many of the hardware systems that appear in practice. Bose and Fisher [2] use similar techniques to verify pipeline circuits with respect to a simpler abstract model by means of a representation function, in analogy to abstract data type verification. By limiting the class of formulas that they handle, these techniques can check certain properties very efficiently. However, these restrictions are also a disadvantage compared to general model checking algorithms. The number of time units that a formula can "look ahead in the future" is bounded by the maximum nesting of $\mathbf{X}$ operators. There is no analog of the *until* operator that can look arbitrarily far into the future. Consequently, the logic is not really suitable for reasoning about nondeterministic systems. For example, at high levels of abstraction, computations are often modeled as taking an arbitrary but finite number

of steps. It is not possible to verify that such a system will make progress using only the **X** operator.

It is difficult to accurately compare the performance of all of the symbolic verification methods. We believe that the best comparison technique is to study how the CPU time required for verification grows asymptotically with larger and larger instances of a class of circuits. For all of the example circuits we have tried with our methods, this growth rate is a small polynomial in the number of components of the circuit. Of the other groups mentioned above, only Bryant, Beatty and Seger [8] have demonstrated good asymptotic performance on a nontrivial class of circuits. Berthet, Coudert and Madre [1] did demonstrate verification times that were sublinear in the number of states in the system, but these times were still exponential in the number of components.

The remainder of the paper is organized as follows. After reviewing BDDs in section 2, we show how to use BDDs to represent circuits in section 3. We describe algorithms for finding reachable states and computing relational products in sections 4 and 5, respectively. Symbolic algorithms for CTL model checking are described in section 6. Empirical results are given for synchronous circuits in section 7 and for asynchronous circuits in section 8. We close with some discussion in section 9.

2 Binary Decision Diagrams

Ordered binary decision diagrams (BDDs) are a canonical form representation for boolean formulas [7]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of CAD applications, including symbolic simulation, verification of combinational logic and, more recently, verification of sequential circuits. A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Consider, for example, the BDD of figure 1. It represents the formula $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of boolean values to the variables a, b, c and d, one can decide whether the assignment makes the formula true by traversing the graph beginning at the root and branching at each node based on the value assigned to the variable that labels the node. For example, the assignment $\langle a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1 \rangle$ leads to a leaf node labeled 1, hence the formula is true for this assignment.

Bryant showed that given a variable ordering, there is a canonical BDD for every formula [7]. The size of the BDD can depend critically on the variable ordering. Bryant gives algorithms for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for formulas f and g. These algorithms have complexity linear in the product of the sizes of the argument BDDs. The only other operations which we require for the algorithms that follow are quantification over boolean variables and substitution of variable names. Bryant gives an algorithm for computing the BDD for a restricted formula of the form $f|_{v=0}$ or $f|_{v=1}$, i.e., f with the variable v set to 0 or 1. The restriction algorithm allows us to compute the BDD for the formula $\exists v[f]$, where v is a boolean variable and f is a formula, as $f|_{v=0} \vee f|_{v=1}$. The substitution of a variable w for a variable v in a formula f, denoted $f\langle v \leftarrow w \rangle$ can be

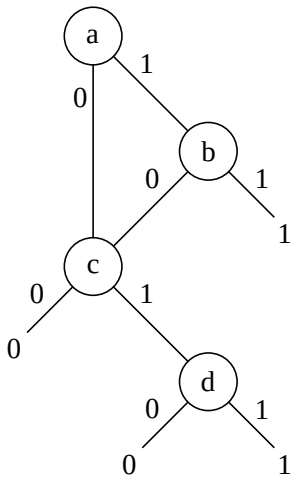

Figure 1: A Binary decision diagram

accomplished using quantification:

$$f\langle v \leftarrow w\rangle = \exists v[(v \Leftrightarrow w) \wedge f].$$

More efficient algorithms are possible, however, for the case of quantification over multiple variables, or multiple renamings. In the latter case, efficiency depends on the ordering of variables in the BDDs being the same on both sides of the substitution.

Another way to view BDDs is as a form of deterministic finite automata. An n-argument boolean function can be identified with the set of strings in $\{0,1\}^n$ that evaluate to 1. Since this is a finite language and all finite languages are regular, there is a minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original boolean function. Logical operations on boolean functions can be implemented by set operations on the languages accepted by the finite automata. For example, AND corresponds to set intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages. The standard BDD operations can be viewed as analogs of these constructions.

3 Representing Circuits

We begin by describing how to represent circuits symbolically. This involves representing sets of circuit states and deriving the transition relation of the circuit. Consider a circuit with a set V of state holding nodes. For a synchronous circuit, the set V is typically the outputs of all the registers in the circuit together with the primary inputs. In the case of an asynchronous circuit, V is usually the set of all nodes. A state of the circuit can be described by giving values for all the nodes in V. Alternatively, if we create a boolean variable for each node in V, then a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can also write a boolean expression which is true for exactly that valuation. For example, given $V = \{v_0, v_1, v_2\}$ and the valuation $\langle v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$,

we derive the boolean formula $v_0 \wedge v_1 \wedge \neg v_2$. This boolean formula can then be represented using a BDD. In general, a boolean formula may be true for many different valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe sets of states by boolean formulas and, hence, by BDDs. In practice, BDDs are often much more efficient than representing sets of states explicitly. We denote sets of states with the letter S, and we denote the BDD representing the set S by $S(V)$, where V is the set of variables that the BDD may depend on.

In addition to representing sets of states of a circuit, we must be able to represent the transitions that the circuit can make. To do this, we extend the idea used above. Instead of just representing a set of states using a BDD, we represent a set of ordered pairs of states. We cannot do this using just a single copy of the state variables, so we create a second set of variables V'. We think of the variables in V as *present state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable in V', which we denote by v'. A valuation for the variables in V and V' can be viewed as designating an ordered pair of states in the circuit, and we can represent sets of these valuations using BDDs as above. We refer to sets of pairs of states as transition relations. If N is a transition relation, then we write $N(V, V')$ to denote the BDD that represents it. We always use an ordering for the BDD variables for which the present and next state variables are interleaved and every present state variable v is adjacent to its corresponding next state variable v'.

3.1 Synchronous Circuits

The method for deriving the transition relation of a synchronous circuit can be illustrated using a small example. The circuit in figure 2 is a modulo 8 counter. Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v'_0, v'_1, v'_2\}$ be another copy of the state variables. The transitions of the modulo 8 counter are given by

$$\begin{aligned} v'_0 &= \neg v_0, \\ v'_1 &= v_0 \oplus v_1, \\ v'_2 &= (v_0 \wedge v_1) \oplus v_2. \end{aligned}$$

The above equations can be used to define the relations

$$\begin{aligned} N_0(V, V') &= (v'_0 \Leftrightarrow \neg v_0), \\ N_1(V, V') &= (v'_1 \Leftrightarrow v_0 \oplus v_1), \\ N_2(V, V') &= (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2), \end{aligned} \tag{1}$$

which describe the constraints each v'_i must satisfy in a legal transition. These constraints can be combined by taking their conjunction to form the transition relation

$$N(V, V') = N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V').$$

In the general case of a synchronous circuit with n state holding nodes, we let $V = \{v_0, \ldots, v_{n-1}\}$ and $V' = \{v'_0, \ldots, v'_{n-1}\}$. Analogous to the modulo 8 counter, for each state variable v'_i there is a function f_i such that

$$v'_i = f_i(V).$$

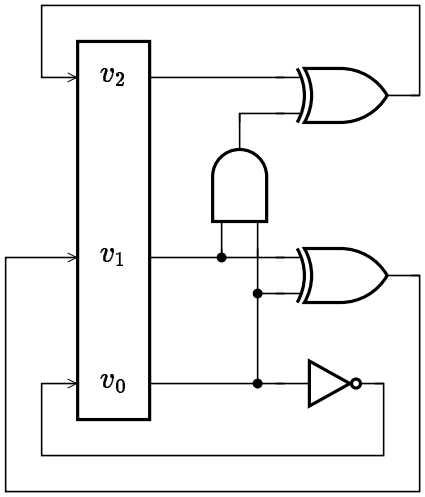

Figure 2: Synchronous modulo 8 counter.

These equations are used to define the relations

$$N_i(V, V') = (v_i' \Leftrightarrow f_i(V)).$$

Continuing the analogy with the modulo 8 counter, the conjunction of these relations forms the transition relation

$$N(V, V') = N_0(V, V') \wedge \cdots \wedge N_{n-1}(V, V').$$

Thus, the transition relation for a synchronous circuit can be expressed as a conjunction of relations.

Given a BDD for each function f_i, it is straightforward to compute the BDD that represents N. We say such a transition relation is *monolithic* because it is represented by a single BDD. Monolithic transition relations have been used successfully for CTL model checking [12], but the primary bottleneck is the size of the BDD for the transition relation.

3.2 Asynchronous Circuits

As with synchronous circuits, the transition relation for an asynchronous circuit can be expressed as a conjunction of relations. Alternatively, it can be expressed as a disjunction. To simplify the description of how these forms of transition relation are obtained, we assume that all the components of the circuit have exactly one output, and have no internal state variables. In this case, it is possible to completely describe each component by a function $f_i(V)$; given values for the present state variables V, the component drives its output to the value specified by $f_i(V)$. For some components, such as C-elements and flip-flops, the function $f_i(V)$ may depend on the current value of the output of the component, as well as the inputs. Extending the method to handle components with multiple outputs is straightforward.

In speed-independent asynchronous circuits, there can be an arbitrary delay between when a transition is enabled and when it actually occurs. We can model this by allowing

each component to nondeterministically choose whether to transition its output. This results in a conjunction of n parts, all of the form

$$N_i(V, V') = \Big(v_i' \Leftrightarrow f_i(V)\Big) \vee (v_i' \Leftrightarrow v_i).$$

The above model for asynchronous circuits allows wires to transition concurrently. We can also use an interleaving model, which allows only one wire to transition at a time. First, we apply the distributive law to the conjunction of the N_i, giving a disjunction of 2^n terms. Each of these terms corresponds to the simultaneous transitioning of some subset of the n wires in the circuit. Second, we keep only those terms that correspond to exactly one wire being allowed to transition. This results in a disjunction of the form

$$N(V, V') = N_0(V, V') \vee \cdots \vee N_{n-1}(V, V'),$$

where

$$N_i(V, V') = \Big(v_i' \Leftrightarrow f_i(V)\Big) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

It is possible for an interleaving model of an asynchronous circuit to give a different set of reachable states than a non-interleaving model. However, this does not occur for the asynchronous circuits we verified in section 8.

3.3 Partitioned Transition Relations

Monolithic transition relations are not the most efficient way to represent the possible transitions of a circuit. Recall that the transition relations for synchronous and asynchronous circuits have the form of conjunctions or disjunctions of a number of pieces $N_i(V, V')$. Each of these pieces can typically be represented by a small BDD. In our experience, these BDDs usually have fewer than 100 nodes, often many fewer; only very rarely do they have more than 1000 nodes. Instead of forming the conjunction or disjunction of the $N_i(V, V')$, we can represent the circuit by a list of these BDDs, which are implicitly conjuncted or disjuncted. We call such a list a *partitioned transition relation* [10, 11].

For the conjunctive transition relations described above, the N_i could be of the form

$$N_i(V, V') = (v_i' \Leftrightarrow f_i(V))$$

for synchronous circuits or

$$N_i(V, V') = \Big(v_i' \Leftrightarrow f_i(V)\Big) \vee (v_i' \Leftrightarrow v_i)$$

for asynchronous circuits, where f_i is a transition function. It can be shown that the size of the BDD for each N_i is at most a constant factor larger than the BDD for f_i. In practice, the difference in size is insignificant. Effectively, using partitioned transition relations to represent a circuit requires no more BDD nodes than using transition functions.

For the disjunctive transition relations described above, the N_i could be of the form

$$N_i(V, V') = \Big(v_i' \Leftrightarrow f_i(V)\Big) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

In this case the BDD for N_i is not guaranteed to be only a constant factor larger than the BDD for f_i; it could be a factor of n larger, where n is the number of state variables of the entire circuit. However, there is an additional technique for efficiently representing relations of this form. Let

$$R(V, V') = v_i' \Leftrightarrow f_i(V).$$

Use the pair $(R(V, V'), i)$ to represent $N_i(V)$ with the interpretation that for all $v_j' \in V'$, if $j = i$ then v_j' is constrained by $R(V, V')$, otherwise v_j' is constrained to be equal to v_j. Our software for manipulating transition relations has been adapted to take advantage of this representation.

While a partitioned transition relation with one BDD for each state variable is almost always more efficient than constructing a monolithic transition relation, it may not be the best choice. As long as the BDDs do not become too large, it is better to combine some of the $N_i(V, V')$ into one BDD by forming their conjunction or disjunction, as appropriate. Fewer BDD nodes may be needed in this representation if the N_i that are combined have similar structure near the root of their BDDs. Combining some of the BDDs in a partitioned transition can also speed up the algorithms for model checking and reachability analysis (see section 5).

4 Finding Reachable States

Many of the ideas used in symbolic verification can be explained by considering the problem of computing reachable state sets, since reachable state computations are at the heart of model checking, state machine comparison, etc. Let S_0 be a set of states, represented by the BDD $S_0(V)$. We wish to compute a BDD $S(V)$ that represents the states reachable from S_0 via the transitions in the transition relation N. We first consider the problem of finding those states S_1 reachable in at most one step from S_0. This set of states is given by

$$S_1 = S_0 \cup \left\{ s' \;\middle|\; \exists s \left[s \in S_0 \wedge (s, s') \in N \right] \right\}.$$

Given the BDDs $S_0(V)$ and $N(V, V')$, we can compute a BDD representing S_1 by performing the logical operations corresponding to the above expression:

$$S_1(V') = S_0(V') \vee \exists_{v \in V} \left[S_0(V) \wedge N(V, V') \right].$$

(The existential quantifier notation above indicates the existential quantification of all variables v in V.) Similarly, those states reachable in at most two steps are represented by

$$S_2(V') = S_0(V') \vee \exists_{v \in V} \left[S_1(V) \wedge N(V, V') \right].$$

In general, the states reachable in at most $k + 1$ steps are represented by

$$S_{k+1}(V') = S_0(V') \vee \exists_{v \in V} \left[S_k(V) \wedge N(V, V') \right].$$

Note that each set of states is a superset of the previous one. Since the total number of states is finite, at some point we must have $S_{k+1} = S_k$. No further states are reachable, so the set of all reachable states is represented by $S(V) = S_k(V)$.

The above computation can be viewed as finding a *least fixed point*. A fixed point of a function f is some value x such that $f(x) = x$. If we have an ordering on values, and x is the smallest fixed point under the order, then x is the least fixed point of f. The *greatest fixed point* is analogously defined. The functions that we will be interested in are functions from sets of states to sets of states. We call such a function a *predicate transformer*. We will use set containment as the ordering between sets of states. A predicate transformer F is *monotonic* if $S \subseteq S'$ implies $F(S) \subseteq F(S')$. A basic result of fixed point theory is that monotonic functions have a well-defined least fixed point and greatest fixed point.

Consider the predicate transformer F defined by

$$F(S) = S_0 \cup \left\{ s' \;\middle|\; \exists s \left[s \in S \wedge (s, s') \in N \right] \right\}.$$

If we represent the state sets by BDDs, the function F can be viewed as specifying a sequence of logical operations on BDDs. In particular,

$$(F(S))(V') = S_0(V') \vee \exists_{v \in V} \left[S(V) \wedge N(V, V') \right]$$

Note that $(F(S_i))(V') = S_{i+1}(V')$. Thus, applying F represents one step in the reachability computation. The sequence of state sets $\emptyset$, $S_0 = F(\emptyset)$, $S_1 = F^2(\emptyset) = F(F(\emptyset))$, etc., converges to the least fixed point of F under the set containment ordering. This least fixed point is exactly the set of reachable states. *Direct iteration* is the method of computing the BDD $S(V')$ representing this fixed point by repeatedly computing $S_{i+1}(V')$ from $S_i(V')$. The predicate transformer F also has a greatest fixed point under set inclusion. This fixed point may also be obtained via direct iteration by starting from the set of all states.

Computing fixed points of predicate transformers similar to F is a fundamental step in symbolic verification, so it is worthwhile to examine the computational complexity of this problem. The direct iteration method involves repeatedly computing $(F(S_i))(V')$ and checking the equivalence of $S_i(V')$ and $S_{i+1}(V')$ in order to determine whether a fixed point has been reached. The time complexity of checking equivalence is either constant or linear in the sizes of the BDDs representing the formulas, depending on the BDD implementation. Most of the computational effort goes into computing $(F(S_i))(V')$. The most expensive step of this is computing

$$\exists_{v \in V} \left[S_i(V) \wedge N(V, V') \right].$$

This is an example of a *relational product* computation. Although relational products can be computed using the normal BDD algorithms for restriction and boolean connectives, it is much more efficient to use a special purpose algorithm. We will discuss this algorithm in section 5.

4.1 Frontier Set Simplification

In order to perform reachability computations more efficiently, a technique called *frontier set simplification* due to Coudert, Berthet, and Madre [18] is often used. Their technique

often reduces the size of the BDD representing the set of states on the "search frontier" (i.e., the set of states in S_{i+1} but not in S_i). Consider the set of states S_2 described above:

$$S_2 = S_0 \cup \left\{ s' \;\middle|\; \exists s \left[s \in S_1 \wedge (s, s') \in N \right] \right\}.$$

Suppose that we step forwards from the states on the search frontier $S_1 - S_0$, i.e., we compute:

$$\left\{ s' \;\middle|\; \exists s \left[s \in (S_1 - S_0) \wedge (s, s') \in N \right] \right\}.$$

This yields a superset of $S_2 - S_1$ (it may also include some states in S_1). If we then add in all the states in S_1, we will obtain S_2. Thus, the expression for S_2 can be rewritten as:

$$S_2 = S_1 \cup \left\{ s' \;\middle|\; \exists s \left[s \in S_1' \wedge (s, s') \in N \right] \right\},$$

where S_1' is the frontier $S_1 - S_0$. In fact, it is sufficient to choose any S_1' satisfying $S_1 - S_0 \subseteq S_1' \subseteq S_1$. Given this freedom, we would like to choose S_1' so that its BDD representation is small. Coudert, Berthet, and Madre describe an algorithm for this. Their procedure takes two BDDs $S(V)$ and $C(V)$ as input: we view these as a state set and a "care set". It produces as output a BDD $S_1'(V)$ such that $S(V) \wedge C(V) = S_1'(V) \wedge C(V)$ (that is, $S(V)$ and $S_1'(V)$ agree for the states that we care about) and such that the BDD $S_1'(V)$ is usually smaller than the BDD $S(V)$. Intuitively, we are simplifying the representation of the set S by adding or removing states not in C. In the example above, we would apply the simplification algorithm to $S_1(V)$ using the complement of S_0 as the "care set".

Using this idea, the algorithm for computing the set of reachable states is modified as follows. First, let S_0' be equal to S_0. The set S_{k+1} of states reachable in $k+1$ or fewer steps is given by

$$S_{k+1} = S_k \cup \left\{ s' \;\middle|\; \exists s \left[s \in S_k' \wedge (s, s') \in N \right] \right\},$$

where S_k' is the result of simplifying the set S_k relative to the "care set" given by the complement of S_{k-1}. Notice that using frontier set simplification does not result in a memory savings; all of the BDDs in the original reachability algorithm are still computed. In fact, memory usage can increase since the BDDs for the S_k' must be computed. The potential advantage of frontier set simplification is that smaller BDDs are used in the relational product computation. In practice, frontier set simplification usually results in an insignificant increase in memory usage and a significant constant factor decrease in computation time.

4.2 Iterative Squaring

One potential problem with reachability computations is that the number of iterations needed to find a fixed point may be exponential in the number of components of the system. We have studied a method for computing fixed points called *iterative squaring* that can drastically reduce the number of iterations needed [12, 13, 14]. The direct iteration algorithm computes the least fixed point of F by computing $F(\emptyset)$, $F^2(\emptyset)$, $F^3(\emptyset)$, etc., until a fixed point is reached. Iterative squaring depends on noting that the predicate transformer F^2, which is given by

$$F^2(S) = S_0 \cup \left\{ s' \;\middle|\; \exists s \left[s \in S \wedge ((s, s') \in N \vee \exists s'' \left[(s, s'') \in N \wedge (s'', s') \in N \right]) \right] \right\},$$

is of the same form as F; the difference is that N has been replaced by

$$N \cup \Big\{ (s,s') \;\Big|\; \exists s''[(s,s'') \in N \wedge (s'',s') \in N] \Big\}.$$

The BDD representation of this relation can be computed as

$$N(V,V') \vee \exists_{v'' \in V''} \Big[N(V,V'') \wedge N(V'',V') \Big].$$

We call this operation *squaring* N. Let N_0 denote N, and let N_{i+1} be the square of N_i. The predicate transformer $F^{(2^i)}$ is equal to

$$F^{(2^i)}(S) = S_0 \cup \Big\{ s' \;\Big|\; \exists s\,[s \in S \wedge (s,s') \in N_i] \Big\}.$$

By repeated squaring starting from N, we eventually reach a fixed point N_k which is the transitive closure of N. Using N_k to compute $F^{(2^k)}(\emptyset)$ gives the least fixed point of F directly. The number of steps needed to compute the fixed point with this method is logarithmic in the number of steps needed with direct iteration (assuming the diameter of the state graph is not reduced when restricted to reachable states). Note, however, that this approach may be impractical if the BDDs needed to represent the intermediate computations become too large. Unfortunately, this appears to be the normal case in practice. In our experience, iterative squaring has been more efficient than direct iteration only on extremely simple examples such as counters.

4.3 Invariants

Invariant checking is a standard method for verifying safety properties of systems. In our context, invariant checking requires defining a set of states W to be an invariant and defining a set Z_0 of "bad" states, which are states that the circuit being verified should not enter. It must then be verified that

1. the initial state (or states) is contained in W,
2. all states reachable from W are contained in W, and
3. W and Z_0 are disjoint.

Clearly these conditions are sufficient for showing that none of the states in Z_0 are reachable.

The exact definition of Z_0 depends on the correctness criteria for the circuit. For an asynchronous circuit, Z_0 might be the set of states in which a hazard can occur. The method can also be used to check whether two synchronous circuits have equivalent input/output behavior. In this case Z_0 is the set of global states (ordered pairs (s_0, s_1) where s_0 and s_1 are each states of the respective circuits) where the two circuits have different outputs.

In verification methods that use theorem provers, the invariant W is typically represented by a formula in some appropriate logic. The user must usually expend a lot of effort to discover the invariant. One of the main advantages of using finite state methods is that an invariant can be constructed automatically without any user intervention. To be more specific, let S_0 be the set of initial states of a circuit, then compute the set S of states

reachable from S_0. If $W = S$, then W clearly satisfies requirements 1 and 2. Once S is computed, it is easy to determine whether the third requirement is satisfied. If the third requirement is not satisfied, then the circuit is incorrect and there does not exist a set of states W that satisfies all three requirements.

An obvious refinement is to check whether any states in Z_0 are reachable while S is being computed, instead of waiting until the computation is complete. Also, it is possible to represent Z_0 as an implicit disjunction of several BDDs, analogous to partitioned transition relations. This can significantly reduce the number of BDD nodes needed to represent Z_0, and does not complicate checking requirement 3. This representation of Z_0 is very helpful when checking for hazards in asynchronous circuits. The set S_0 can also be represented by an implicit disjunction of BDDs, but this requires computing a different set of reachable states for each of the BDDs used, and is not helpful for typical sets of initial states.

In some cases, however, computing the states reachable from the initial states is too inefficient. For example, consider verifying a circuit that contains a n bit counter that is incremented every cycle. If the counter is set to the same value in all of the initial states, then computing S will require at least $O(2^n)$ steps. We describe two methods, which do not involve iterative squaring, for speeding up the computation of an invariant.

The first method involves computing an invariant from Z_0 rather than from S_0. Compute, by reverse reachability analysis, the set Z of all states from which some state in Z_0 can be reached. Thus, Z is the set of all states that can reach a "bad" state. If W is the complement of Z, then W satisfies requirements 2 and 3. The circuit is correct if and only if W also satisfies the first requirement, which is equivalent to $S_0 \cap Z = \emptyset$. Thus, the difference between forward and reverse reachability analysis is that the transition relation is reversed, and the roles of S_0 and Z_0 are swapped. Reverse reachability analysis has been studied by Filkorn [24], and it can be viewed as a generalization of a earlier methods for finding equivalent states in finite state machines [26, 32].

In some cases, an invariant can be computed much more quickly with reverse reachability analysis than forward, even if both methods compute the same invariant. As an extreme example, consider using reachability analysis to verify that two identical n bit counters have the same input/output behavior. The set Z_0 is the set of states where the two counters have different outputs. If S_0 is a singleton set, then 2^n steps will be required to compute S. However, Z can be computed very quickly since $Z = Z_0$ in this case.

For the second method for speeding up the computation of invariants, notice that automatic computation of invariants and user construction of invariants are just two ends of a continuum. We will only describe the forward reachability case here, but the idea can also be applied in reverse using the duality described above. The user must choose a set T_0 of states such that $S_0 \subseteq T_0$; then the set T of states reachable from T_0 is computed. If T_0 is chosen well, then T can be computed from T_0 more quickly (in fewer iterations) than S can be computed from S_0. Usually $T_0 \subseteq S$, in which case $T = S$. However, this need not be the case. All that is required for the verification to go through is that $T \cap Z_0 = \emptyset$. We use this idea in the verification of an asynchronous stack circuit (see section 8.2). Rather than starting the reachability search from the set S_0 of initial states (where the stack is empty), we set T_0 to the set of all possible quiescent states of the stack. This significantly reduces the number of iterations necessary to reach a fixed point.

5 Computing Relational Products

As noted earlier, computing relational products is a fundamental operation in many symbolic verification methods. This section describes the techniques that we use for relational product computations.

5.1 Basic Algorithm

Consider the following relational product:

$$S'(V') = \exists_{v \in V} \left[S(V) \wedge N(V, V') \right].$$

In figure 3, we give a special BDD algorithm *RelProd* that performs this computation in one pass over the BDDs $S(V)$ and $N(V, V')$. This is important in practice since the relational product is computed without ever constructing the BDD for

$$S(V) \wedge N(V, V'),$$

which is often fairly large. The basic idea behind the algorithm is to perform the normal conjunction, except that every time we would build a node labeled with an element of V, we perform a disjunction. The BDD $S'(V')$ is computed with the call $\mathit{RelProd}(S(V), N(V, V'), V)$.

Like many BDD algorithms, *RelProd* uses a result cache. In this case, entries in the cache are of the form (f, g, E, h), where E is a set of variables and f, g and h are BDDs. If such an entry is in the cache, it means that a previous call to $\mathit{RelProd}(f, g, E)$ returned h as its result.

The algorithm as presented is independent of assumptions about the BDD variable ordering. In our implementation, it has been optimized for the case where the present state and next state variables are interleaved, with corresponding present and next state variables adjacent to each other.

The algorithm, while working well in practice (assuming $N(V, V')$ is reasonably sized), has exponential complexity in the worst case. Most of the situations where this complexity is observed are cases in which the output $S'(V')$ is exponentially larger than the inputs $S(V)$ and $N(V, V')$. In such situations, any method of computing $S'(V')$ must have exponential complexity.

The basic relational product algorithm requires having $N(V, V')$ be a monolithic transition relation, consisting of a single BDD. We saw in section 3 how to construct this BDD for synchronous and asynchronous circuits. Unfortunately, for many practical examples, this BDD is very large. Partitioned transition relations can provide a much more concise representation, but they cannot be used with the basic relational product algorithm. In the next two subsections, we show how to extend the basic algorithm to compute relational products for partitioned transition relations.

```
function RelProd(f, g: BDD, E: set of variables): BDD
if f = false ∨ g = false
    return false
else if f = true ∧ g = true
    return true
else if (f, g, E, h) is in the result cache
    return h
else
    let x be the top variable of f
    let y be the top variable of g
    let z be the topmost of x and y
    h_0 := RelProd(f|_{z=0}, g|_{z=0}, E)
    h_1 := RelProd(f|_{z=1}, g|_{z=1}, E)
    if z ∈ E
        h := Or(h_0, h_1)
             /* BDD for h_0 ∨ h_1 */
    else
        h := IfThenElse(z, h_1, h_0)
             /* BDD for (z ∧ h_1) ∨ (¬z ∧ h_0) */
    endif
    insert (f, g, E, h) in the result cache
    return h
endif
```

Figure 3: Relational product algorithm

5.2 Disjunctive Partitioning

For a disjunctive partitioned transition relation, the relational product computed is of the form

$$S'(V') = \exists_{v \in V} \Big[S(V) \wedge (N_0(V,V') \vee \cdots \vee N_{n-1}(V,V')) \Big].$$

This relational product can be computed without ever constructing the BDD for the full transition relation by distributing the existential quantification over the disjunctions:

$$S'(V') = \exists_{v \in V} \Big[S(V) \wedge N_0(V,V') \Big] \vee \cdots \vee \exists_{v \in V} \Big[S(V) \wedge N_{n-1}(V,V') \Big].$$

Thus, we are able to reduce the problem of computing $S'(V')$ to one of computing a series of relational products involving relatively small BDDs. Much larger asynchronous circuits can be verified using this representation than with a monolithic transition relation.

5.3 Conjunctive Partitioning

When using a conjunctive partitioned transition relation, the relational product computed is of the form

$$S'(V') = \exists_{v \in V} \Big[S(V) \wedge (N_0(V,V') \wedge \cdots \wedge N_{n-1}(V,V')) \Big]. \tag{2}$$

The main difficulty in computing $S'(V')$ without building the conjunction is that existential quantification does not distribute over conjunction. The method given below overcomes this difficulty.

Our technique [10, 11] is based on two observations. First, circuits exhibit locality, so many of the $N_i(V,V')$ will depend on only a small number of the variables in V and V'. Second, although existential quantification does not distribute over conjunction, subformulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. We will take advantage of these observations by conjuncting the $N_i(V,V')$ with $S(V)$ one at a time and using "early quantification" to quantify out each variable v when none of the remaining $N_i(V,V')$ depend on v.

Consider the modulo 8 counter described in section 3.1. In this case,

$$S'(V') = \exists v_0 \, \exists v_1 \, \exists v_2 \Big[S(V) \wedge (N_0(V,V') \wedge N_1(V,V') \wedge N_2(V,V')) \Big].$$

Since conjunction is commutative and associative, we can rewrite this as

$$S'(V') = \exists v_0 \, \exists v_1 \, \exists v_2 \Big[((S(V) \wedge N_2(V,V')) \wedge N_1(V,V')) \wedge N_0(V,V') \Big]. \tag{3}$$

The reasons for computing the conjunctions in this particular order will become clear momentarily. As mentioned above, subformulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. According to equation 1, $N_0(V,V')$ does not depend on v_1 or v_2; thus,

$$S'(V') = \exists v_0 \Big[\exists v_1 \, \exists v_2 \Big[(S(V) \wedge N_2(V,V')) \wedge N_1(V,V') \Big] \wedge N_0(V,V') \Big].$$

Since $N_1(V, V')$ does not depend on v_2, we can apply this trick one more time by writing

$$S'(V') = \exists v_0 \left[\exists v_1 \left[\exists v_2 \left[(S(V) \wedge N_2(V, V')\right] \wedge N_1(V, V')\right] \wedge N_0(V, V')\right].$$

We can now compute the relational product in equation 2 by starting with $S(V)$ and at each step combining the previous result with an $N_i(V, V')$ and quantifying out the appropriate variables. Thus, we have reduced the problem of computing the full relational product to one of performing a series of smaller relational product-like steps. Notice that the intermediate results may depend both on variables in V and variables in V'.

Now we can explain why we chose the ordering of conjuncts given in equation 3. We wish to order the $N_i(V, V')$ so that the variables in V can be quantified out as soon as possible and the variables in V' are added as slowly as possible. This is desirable since it reduces the number of variables that the intermediate BDDs depend on and hence can greatly reduce the size of these BDDs. In this particular example, the variables in V' are added one at a time, independent of the ordering of the $N_i(V, V')$. Thus, the optimum ordering for the $N_i(V, V')$ is determined by how quickly the variables in V can be quantified out. For each of the variables v_i in V, consider the number of terms that depend on v_i: all 4 terms depend on v_0, while 3 terms depend on v_1, and 2 terms depend on v_2. Thus, v_2 is the best candidate for a variable to quantify out early. This explains why we chose to combine $S(V)$ and $N_2(V, V')$, the two terms that depend on v_2, as the first step in the computation. Similarly, $N_1(V, V')$ was chosen next because it was the only remaining term that depended on v_1.

The above example involved computing the relational product in a forward reachability search. Computing relational products for backward reachability search is quite similar to the forward reachability case described above. However, instead of quantifying out the present state variables when performing the relational product, we quantify out the next state variables. This change may affect the optimal ordering of the $N_i(V, V')$ when using conjunctive partitioning. To illustrate this, we consider the modulo 8 counter again. The relational product that we want to compute has the form

$$S'(V) = \exists v_0' \exists v_1' \exists v_2' \left[S(V') \wedge (N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V'))\right].$$

We rewrite this as

$$S'(V) = \exists v_2' \exists v_1' \exists v_0' \left[((S(V') \wedge N_0(V, V')) \wedge N_1(V, V')) \wedge N_2(V, V')\right]. \tag{4}$$

Since, according to equation 1, $N_2(V, V')$ does not depend on v_0' or v_1',

$$S'(V) = \exists v_2' \left[\exists v_1' \exists v_0' \left[(S(V') \wedge N_0(V, V')) \wedge N_1(V, V')\right] \wedge N_2(V, V')\right].$$

Since $N_1(V, V')$ does not depend on v_0', we obtain

$$S'(V) = \exists v_2' \left[\exists v_1' \left[\exists v_0' \left[(S(V') \wedge N_0(V, V')\right] \wedge N_1(V, V')\right] \wedge N_2(V, V')\right].$$

In this particular example, the number of new state variables v_i' in the intermediate BDDs is independent of the ordering of the $N_i(V, V')$. However, the number of old state variables v_i depends on the ordering, and is minimized by the ordering given in equation 4. Note that this ordering is different from the one in equation 3.

The method described above for computing the relational product for the modulo 8 counter can be generalized to an arbitrary conjunctive partitioned transition relation with n state variables, as follows. The user must choose a permutation ρ of $\{0, \ldots, n-1\}$. This permutation determines the order in which the partitions $N_i(V, V')$ are combined. For each i, let D_i be the set of variables in V that $N_i(V, V')$ depends on. Also, let

$$E_i = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}.$$

Thus, E_i is the set of variables contained in $D_{\rho(i)}$ that are not contained in $D_{\rho(k)}$ for any k larger than i. The E_i are pairwise disjoint and their union is equal to V. The relational product in equation 2 can be computed as

$$\begin{aligned}
S_1(V, V') &= \exists_{v \in E_0} \left[S(V) \wedge N_{\rho(0)}(V, V') \right] \\
S_2(V, V') &= \exists_{v \in E_1} \left[S_1(V, V') \wedge N_{\rho(1)}(V, V') \right] \\
&\vdots \\
S'(V') &= \exists_{v \in E_{n-1}} \left[S_{n-1}(V, V') \wedge N_{\rho(n-1)}(V, V') \right].
\end{aligned}$$

The ordering ρ has a significant impact on how early in the computation state variables can be quantified out. This affects the size of the BDDs constructed and the efficiency of the verification procedure. Thus, it is important to choose ρ carefully, just as with the BDD variable ordering. In practice, we have found it fairly easy to come up with orderings which give good results.

5.4 Recombining Partitions

Earlier, we described how a circuit could be represented by a set of transition relations $N_i(V, V')$, each depending on exactly one variable in V'. We also pointed out that combining some of the N_i together into one BDD can result in a smaller representation. Combining parts of a transition relation in this way can also significantly speed up the computation of relational products.

For example, consider the case of an n bit counter. With the usual variable ordering, the number of BDD nodes needed to represent the transition relation is linear in n in both the monolithic and fully partitioned cases. Suppose $S(V)$ represents a singleton state set of the counter. Computing $S'(V')$ with the fully partitioned representation requires n BDD operations, each of which has complexity $O(n)$, for a total complexity of $O(n^2)$. On the other hand, if we use the monolithic relation, we perform one operation of complexity $O(n)$, a savings in time of a factor of n. In practice, we can often get a speed up by combining all of the BDDs for any given register, without significantly increasing the number of BDD nodes in the transition relation.

The empirical results in sections 7 and 8 also illustrate the benefits of recombining partitions. In particular, for the KEY benchmark (section 7.2), recombining gave a factor of

25 speed up. The MINMAX example shows how recombining can give a major reduction in the space needed for the transition relation, as well as a significant speed up.

6 Symbolic Model Checking

6.1 Computation Tree Logic

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL or Computation Tree Logic [17]. In this logic each of the usual forward-time operators of linear temporal logic (**G** *globally* or *invariantly*, **F** *sometime in the future*, **X** *next time* and **U** *until*) must be directly preceded by a *path quantifier*. The path quantifier can either be an **A** (for all computation paths) or an **E** (for some computation path). Thus, some typical CTL operators are $\mathbf{AG}f$, which will hold in a state provided that f holds at all points (globally) along all possible computation paths starting from that state, and $\mathbf{EF}f$, which will hold in a state provided that there is a computation path such that f holds at some point in the future on the path.

For explaining our verification procedure, it is convenient to express the CTL operators with universal path quantifiers in terms of the operators with existential path quantifiers, taking advantage of the duality between universal and existential quantification. Consequently, in our description of the syntax and semantics of CTL, we specify the existential path quantifiers directly and treat the universal path quantifiers as syntactic abbreviations.

CTL formulas are constructed from *atomic propositions* using boolean connectives and temporal operators. When verifying a circuit, the set of atomic propositions is typically equal to the set V of state variables of the circuit. If v is an atomic proposition in V, then the formula v is true of a circuit state if and only if the state variable v is high in that state. The formal syntax of CTL formulas is given by the following two rules:

1. every atomic proposition v in V is a formula in CTL; and
2. if f and g are CTL formulas, then so are $\neg f$, $f \vee g$, $\mathbf{EX}f$, $\mathbf{E}[f\mathbf{U}g]$ and $\mathbf{EG}f$.

Let S_0 be the set of initial states of a circuit, and let N be a transition relation. We now define the semantics of CTL for such a system. A *path* from the state s_0 is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $N(s_i, s_{i+1})$ holds for every i. The propositional connectives $\neg$ and $\vee$ have their usual meanings of negation and disjunction. The other propositional operators can be defined in terms of these. **X** is the *next time* operator: $\mathbf{EX}f$ will be true in a state s_0 if and only if there is a path $s_0 s_1 \ldots$ from s_0 such that f is true at s_1. **U** is the *until* operator: $\mathbf{E}[f\mathbf{U}g]$ will be true in a state s_0 if and only if there exists a path $s_0 s_1 \ldots$ from s_0 such that g holds at some s_j and f holds at all s_i for which $i < j$. The operator **G** is used to express the *invariance* of some property over time: $\mathbf{EG}f$ will be true at a state s_0 if there is a path $s_0 s_1 \ldots$ from s_0 such that f holds at each state on the path. If f is true in state s, we say that s satisfies f and write $s \models f$. We say that the system satisfies f if $s \models f$ for all states s in S_0. We will identify a CTL formula f with the set $\{ s \mid s \models f \}$ of states that make f true. We also use the following syntactic abbreviations for CTL formulas:

- $\mathbf{AX}f \equiv \neg\mathbf{EX}\neg f$ which means that f holds at all successor states of the current state (f must hold at every *next* state).

- $\mathbf{EF}f \equiv \mathbf{E}[true\mathbf{U}f]$ which means that for some path, there exists a state on the path at which f holds (f is *possible* in the future).
- $\mathbf{AF}f \equiv \neg\mathbf{EG}\neg f$ which means that for every path, there exists a state on the path at which f holds (f is *inevitable* in the future).
- $\mathbf{AG}f \equiv \neg\mathbf{EF}\neg f$ which means that for every path, at every node on the path f holds (f holds *invariantly* along all paths).
- $\mathbf{A}[f\mathbf{U}g] \equiv \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g$ which means that for every path, there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix (f holds *until* g holds, along all paths).

6.2 Model Checking

Model checking is the problem of determining whether a given CTL formula f is true in a given state transition graph. There is a program called EMC (Extended Model Checker) that verifies the truth of a formula in a model by using efficient graph-traversal techniques. If the model is represented as a state transition graph, the complexity of the algorithm is linear in the size of the graph and in the length of the formula. The algorithm is quite fast in practice [5, 17]. However, an explosion in the size of the model may occur when the state transition graph is extracted from a circuit, particularly if the circuit contains many registers or other memory elements.

In this section, we present a model checking algorithm for CTL which uses BDDs as its internal representation, in order to avoid explicitly enumerating the states of the model. We call this *symbolic model checking*. The algorithm is defined by a procedure CHECK that takes the CTL formula to be checked as its argument. It returns a BDD $S(V)$ that represents exactly those states of the system that satisfy the formula. Of course, the output of CHECK depends on the system being checked; this parameter is implicit in the discussion below. The set S_0 of initial states is represented by a BDD $S_0(V)$, and the transition relation N is represented by the BDD $N(V, V')$ as discussed earlier. We assume that N is *total*, that is, every state has some successor state. This is true for transition relations of the forms described in section 3.

We define CHECK inductively over the structure of CTL formulas. If f is an atomic proposition v, then CHECK(f) is simply the BDD v. The inductive steps for formulas of the form $\mathbf{EX}f$, $\mathbf{E}[f\mathbf{U}g]$, and $\mathbf{EG}f$ are given in terms of intermediate procedures:

$$\begin{aligned}
&\textsc{Check}(\mathbf{EX}f) = \textsc{CheckEX}(\textsc{Check}(f)),\\
&\textsc{Check}(\mathbf{E}[f\mathbf{U}g]) = \textsc{CheckEU}(\textsc{Check}(f), \textsc{Check}(g)),\\
&\textsc{Check}(\mathbf{EG}f) = \textsc{CheckEG}(\textsc{Check}(f)).
\end{aligned}$$

The definitions of these intermediate procedures are given below. Notice that these intermediate procedures take boolean formulas as their arguments, while CHECK takes a CTL formula as its argument. The cases of CTL formulas of the form $f \vee g$ or $\neg f$ are handled using the standard algorithms for computing boolean connectives with BDDs. Since $\mathbf{AX}f$, $\mathbf{A}[f\mathbf{U}g]$ and $\mathbf{AG}f$ can all be rewritten using just the above operators, this definition of CHECK covers all CTL formulas.

The formula $\mathbf{EX}f$ is true in a state if and only if there exists a path from that state for which the second state on the path satisfies f. Since we have assumed that the transition relation is total, this is equivalent to there being a successor of the state which satisfies f. Thus, we define CHECKEX such that

$$\text{CHECKEX}(S(V)) = \exists_{v' \in V'} \Big[S(V') \wedge N(V, V') \Big].$$

Compare the definition of CHECKEX to the relational product in the definition of S_{k+1} in Section 4. They are quite similar except that first case computes the set of states from which a state in S can be reached, while the second computes the states that can be reached from a state in S_k. In other words, CHECKEX performs one step of a backward reachability search instead of a forward reachability search. The techniques described in section 5 for computing relational products can be used here. (However, as discussed in subsection 5.3, we may wish to use different partition orderings for the forward and backward reachability computations when using conjunctive partitioning.)

Recall that the formula $\mathbf{E}[f\mathbf{U}g]$ means that there is a computation beginning in the current state in which g is true in some future state s, and f is true in all the states preceding s. This means that either g is true in the current state, or f is true in the current state and there exists a successor state in which $\mathbf{E}[f\mathbf{U}g]$ is true. In other words, it is the least fixed point of the predicate transformer defined by

$$(F(S))(V) = S_g(V) \vee \Big(S_f(V) \wedge \text{CHECKEX}(S(V)) \Big),$$

where S_g and S_f are the sets of states satisfying g and f, respectively [16]. The algorithm for CHECKEU works by finding the least fixed point of the above predicate transformer. This fixed point can be computed with either the direct iteration or iterative squaring methods described earlier.

The formula $\mathbf{EG}f$ states that there exists a computation beginning with the current state in which f is globally (invariantly) true. This means that f is true in the current state, and $\mathbf{EG}f$ is true in some successor state. This condition is the greatest fixed point of the predicate transformer

$$(F(S))(V) = S_f(V) \wedge \text{CHECKEX}(S(V)),$$

where S_f is the set of states satisfying f. CHECKEG computes this fixed point, either by direct iteration or iterative squaring.

After determining the set S of states that satisfy the formula f, the algorithm checks whether S_0 is a subset of S (that is, whether $\neg S_0(V) \vee S(V)$ is the BDD representing *true*). If it is, then the system satisfies f.

6.3 Fairness Constraints

Next, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not

ignore one of its request inputs forever. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of CTL slightly. A *fairness constraint* can be an arbitrary formula of the logic. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The path quantifiers in CTL formulas are then restricted to fair paths. In the remainder of this section we describe how to modify the algorithm above to handle fairness constraints. We assume the fairness constraints are given by a set of CTL formulas $H = \{h_1, \ldots, h_n\}$.

We define a new procedure CHECKFAIR for checking CTL formulas relative to the fairness constraints in H. We do this by giving definitions for new intermediate procedures CHECKFAIREX, CHECKFAIREU, and CHECKFAIREG which correspond to the intermediate procedures used to define CHECK.

Consider the formula $\mathbf{EG}f$ given fairness constraints H. The formula means that there exists a path beginning with the current state in which f holds globally (invariantly) and each formula in H holds infinitely often. The set of such states S is the largest set with the following two properties:

1. all of the states in S satisfy f, and
2. for all fairness constraints $h_k \in H$ and all states $s \in S$, there is a sequence of states of length one or greater from s to a state in S satisfying h_k such that all states on the path satisfy f.

It is easy to show that if these conditions hold, each state in the set is the beginning of an infinite computational path on which f is always true, and for which every formula in H holds infinitely often. Thus, CHECKFAIREG will compute the greatest fixed point of the predicate transformer given by

$$(F(S))(V) = S_f(V) \wedge \bigwedge_{k=1}^{n} \textsc{CheckEX}(\textsc{CheckEU}(S_f(V), S(V) \wedge \textsc{Check}(h_k))),$$

where S_f is the sets of states satisfying f under the fairness constraints H. The fixed point can be evaluated in the same manner as before. The main difference is that each time the above expression is evaluated, several nested fixed point computations are done (inside CHECKEU).

Checking $\mathbf{EX}f$ and $\mathbf{E}[f\mathbf{U}g]$ under fairness constraints is simpler. The set of all states which are the start of some fair computation is

$$\mathit{fair} = \textsc{CheckFair}(\mathbf{EG}\,\mathit{true}).$$

The formula $\mathbf{EX}f$ is true under fairness constraints in a state s if and only if there is a successor state s' of s such that s' satisfies f and s' is at the beginning of some fair computation path. Thus, the formula $\mathbf{EX}f$ (under fairness constraints) is equivalent to the formula $\mathbf{EX}(f \wedge \mathit{fair})$ (without fairness constraints). Therefore, we define

$$\textsc{CheckFairEX}(S_f(V)) = \textsc{CheckEX}(S_f(V) \wedge \mathit{fair}(V)).$$

Similarly, the formula $\mathbf{E}[f\mathbf{U}g]$ (under fairness constraints) is equivalent to the formula $\mathbf{E}[f\mathbf{U}(g \wedge \mathit{fair})]$ (without fairness constraints). Therefore, we define

$$\textsc{CheckFairEU}(S_f(V), S_g(V)) = \textsc{CheckEU}(S_f(V), S_g(V) \wedge \mathit{fair}(V)).$$

7 Verifying Synchronous Circuits

This section gives empirical results for verifying synchronous circuits using both CTL model checking and reachability analysis. We begin by applying model checking to a simple pipeline circuit. Reachability analysis is applied to two standard benchmark circuits, MINMAX and KEY.

7.1 Pipelined ALU

The pipeline considered in this section performs three-address arithmetic and logical operations on operands stored in a register file. The circuit is a generalized version of one described in an earlier conference paper [12]. Figure 4 shows a block diagram for the pipeline. The number of pipe registers can be varied; if s is the number of pipe registers, then executing an instruction requires $s + 2$ cycles.

1. During the first cycle of the instruction, operands are read from the register file into the instruction operand registers.

2. During the second cycle, the result of the operation is computed and stored in the first pipe register.

3. In cycles three through $s + 1$, the result is passed between pipe registers.

4. In the last cycle, the result is written back to the register file.

We have included extra pipe registers in this version of pipeline to test how the performance of the model checker depends on the number of pipe registers. In a real circuit, operations would typically be performed between some of the pairs of pipe registers, but in our example, results are just propagated unchanged.

Each instruction specifies the source and destination registers and the operation to perform. In addition, the pipeline has a *stall* input that indicates that the instruction is invalid and should be ignored. More specifically, the instruction's destination register should not be affected if the *stall* input is true. The *stall* signal might, for example, be used to indicate an instruction cache miss; the signal would be asserted until an instruction is fetched from main memory. In order to allow results to be used before they are actually written into the register file, data can be fed from the ALU output or from one of the pipe registers back to the ALU operand registers. We experimented with a number of versions of the pipeline with varying numbers of registers r, register widths w, numbers of pipe stages s, and numbers of operations o.

The specification of the pipeline is given in CTL. For simplicity of exposition, we fix the number of general registers r at 2 and the number of pipe registers s at 1, and we assume that the pipeline does only exclusive-or operations. In the actual verification, we used more complex circuits with more operations. The specification that we used consists of two parts. The first specifies that the destination register is updated correctly. This is described by a set of formulas of the following form:

$$\mathbf{AG}\Big(\neg \mathit{stall} \rightarrow ((\mathit{src1op}_i \oplus \mathit{src2op}_i) \Leftrightarrow \mathit{result}_i)\Big).$$

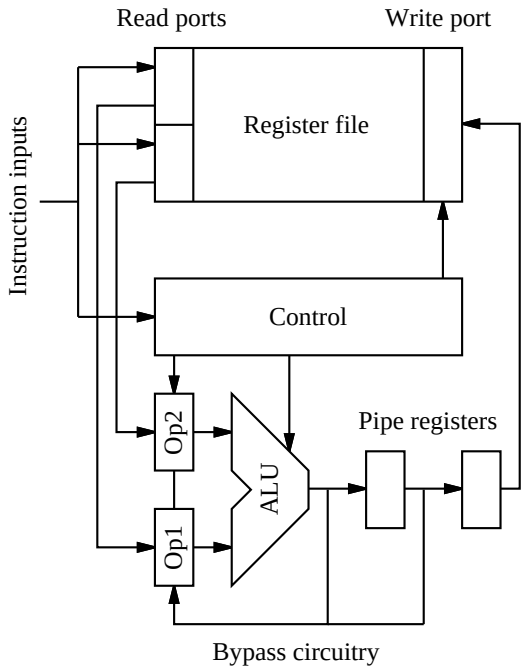

Figure 4: Pipeline circuit block diagram

Here, $src1op_i$ and $src2op_i$ are abbreviations for formulas that represent the value of the ith bit of the two source operands and $result_i$ is a formula that represents the ith bit of the result written into the register file. The overall formula states "if the pipeline is not being stalled, then the ith bit of the result of the current operation should be the exclusive-or of the ith bits of the two source operands".

In order to express $src1op_i$, $src2op_i$ and $result_i$, we need a way of expressing the value stored in a bit of a register some number of cycles k in the future. Since the only nondeterminism in the circuit is input nondeterminism, and since the inputs cannot affect the state of the register file until 3 cycles the future (assuming s equals 1), this can be done using the CTL **AX** operator. That is

$$\underbrace{\mathbf{AXAX}\ldots\mathbf{AX}}_{k} reg_{j,i},$$

which we abbreviate as $\mathbf{AX}^k reg_{j,i}$, represents the value of bit i of register j in k cycles, provided $k \leq 3$. We can check the assumption that the inputs do not affect the register file state before 3 cycles elapse by verifying that $\mathbf{EX}^k reg_{j,i}$ and $\mathbf{AX}^k reg_{j,i}$ are equivalent for k up to 3. The timings given below do not include this check; it increases the times by a factor of two. Now $src1op_i$ is either $\mathbf{AX}^2 reg_{0,i}$ or $\mathbf{AX}^2 reg_{1,i}$, depending on whether the first source address is 0 or 1. The $\mathbf{AX}^2$ accounts for the pipeline latency; in 2 cycles, all the values currently being computed will have been written back into the register file. Thus, we obtain

$$src1op_i = (\neg src1addr_0 \wedge \mathbf{AX}^2 reg_{0,i}) \vee (src1addr_0 \wedge \mathbf{AX}^2 reg_{1,i}).$$

Here, $src1addr_i$ is the ith bit of the first source address input. The formula for $src2op_i$ is analogous. The formula for $result_i$ is also similar, except we use the values in the register

file in 3 cycles (after the operation is completed), and we select based on *destaddr*, the destination address register:

$$result_i = (\neg destaddr_0 \wedge \mathbf{AX}^3 reg_{0,i}) \vee (destaddr_0 \wedge \mathbf{AX}^3 reg_{1,i}).$$

The other part of the specification describes what happens to the registers not being written (or to all the registers when the pipeline stalls). In particular, the register should not be altered by the current operation. For example, for register 1:

$$\mathbf{AG}\Big((stall \vee \neg destaddr_0) \rightarrow (\mathbf{AX}^2 reg_{1,i} \Leftrightarrow \mathbf{AX}^3 reg_{1,i})\Big).$$

Note that a number of common subformulas, such as the formulas $\mathbf{AX}^k reg_{j,i}$, appear throughout the specification. In the experiments described below, the set of states satisfying each of these subformulas was computed only once and then saved.

We performed most of the experiments using a partitioned transition relation to represent the circuit. From the block diagram, we see that the circuit decomposes naturally into pieces. We used this decomposition as a starting point for breaking the transition relation into parts. Some of the parts, such as the register file, were found to require large BDDs to represent; we broke these into more pieces. We also found that we could combine some of the parts, such as most of the pipe registers, without increasing the number of BDD nodes required; we did this to decrease overhead. The final decomposition had the following pieces:

1. control logic;
2. the first pipe register;
3. the other pipe registers;
4. the first ALU operand register;
5. the second ALU operand register; and
6. one piece for each general register.

The ordering above was also the ordering used for processing the transition relation. With this ordering, the number of variables in intermediate results never exceeded the number of state variables by more than w, the register width. We found that the sizes of the intermediate results with this ordering increased monotonically during each step; thus, breaking the transition relation into pieces did not result in having to manipulate larger state set BDDs than would have been necessary with a single monolithic BDD representing the transition relation. This is an important point; in many applications involving BDDs, it is the number of nodes in intermediate results (not the final result) that limits the size of the problem that can be handled.

In the BDD variable ordering that we used, the source address registers are nearest to the root. The bits of these registers are interleaved. These are followed by variables which make up the destination address shift chain (this is a chain of shift registers that are used to hold the destination address for an operation until the result of the operation is written back

into the register file). For each stage in the chain, starting with the leftmost (input) stage, there is a stall bit followed by a destination address register. Next come the two opcode shift registers, with their bits interleaved. The operand registers, general registers and pipe registers, interleaved, are at the end of the ordering. All registers are arranged with most significant bit closest to the root of the BDD, since this results in smaller BDDs for the operations used.

As mentioned, we experimented with a number of versions of the pipeline with varying numbers of registers r, register widths w, numbers of pipe stages s, and numbers of operations o. For each version, we collected information on the sizes of the BDDs needed to represent the transition relation and state sets, and on the time required to do the verification. The following table shows the rate of growth in the sizes of the various pieces of the transition relation as a function of the parameters. These rates of growth were found by studying "profiles" of the BDDs (histograms of the number of nodes labeled with each variable). By considering the circuit's operation and examining how the profiles changed as the parameters varied, we were able to exactly account for the structure of the BDDs needed to represent the transition relation.

control logic	$O(sr \log r)$
pipe registers	$O(ws)$
ALU operand registers	$O\Big(sr \log r + w(r+s)\Big)$
each general register	$O(w + \log r)$

The total number of BDD nodes needed to represent the transition relation grows linearly with each parameter except r, for which it grows at a rate of $r \log r$. The $\log r$ factors arise because an extra addressing bit is needed when r increases from $2^i - 1$ to 2^i. The number of partitions in the transition relation increased linearly with r, and did not depend on w, s or o. The number of BDD nodes in each piece of the transition relation was typically between 10 and 500. No piece ever had more than 1,500 nodes. The way the sizes of the pipe registers and ALU operand registers vary with o depends on the exact operations. The ones we used were addition, subtraction, and bitwise logical operations. With this set, the control logic grew $O(\log o)$, the pipe registers and ALU operand registers grew $O(o)$, and the general registers did not vary with o.

To make it clear how the above bounds on BDD sizes are derived, we consider one specific example: the transition relation for the control logic. The other pieces of the transition relation for the pipelined ALU were analyzed in a similar way. The control logic consists of two parts: the opcode shift chain and the destination address register shift chain. Each shift chain is used to store information about an operation until the time that it is to be used. The opcode is delayed for one cycle while the ALU operand registers are being loaded, and hence the opcode shift chain is described by the following transition relation:

$$\bigwedge_{i=0}^{\lceil \log o \rceil - 1} opcode'_{1,i} \Leftrightarrow opcode_{0,i}.$$

Here, $opcode_{0,i}$ is the ith bit of the input opcode, and $opcode'_{1,i}$ is the (next state value of the) ith bit of the register used to control the ALU. With the variable ordering described

above, this transition relation requires $O(\log o)$ BDD nodes to represent. The destination address register shift chain is used to hold the destination register number until the result of the operation reaches the end of the pipeline. Then the last register in the chain is used to control the writeback into the register file. The transition relation that describes the shift chain is:

$$\bigwedge_{i=1}^{s+1} \bigwedge_{j=0}^{\lceil \log r \rceil - 1} dest'_{i,j} \Leftrightarrow dest_{i-1,j}.$$

In this expression, $dest_{0,i}$ is the ith bit of the destination input. Because of the variable ordering used, the BDD for this transition relation consists of $s+2$ sections, one for each $dest_i$. At the end of each section, the BDD has $O(r)$ width, since the value of $dest_i$ must be "remembered" in order to check that $dest'_{i+1}$ is correct. In addition, each bit of $dest_i$ is "forgotten" before encountering the corresponding bit of $dest_{i+1}$. Hence the width of the BDD is in fact $O(r)$ everywhere. The number of variables that this part of the BDD depends on is bounded by $2(s+1)\lceil \log r \rceil$ (the factor of two accounts for current and next state variables), and hence, the total BDD size is $O(sr \log r)$. The conjunction of the BDDs for the first and second parts gives a BDD of size $O(sr \log r + \log o)$.

We also studied the BDDs representing the various state sets in the verification and used profiles to determine their rates of growth. Since most of the time and space for each verification was used computing and representing the value of the destination register at the end of the current operation, we concentrated on these. Again, by understanding the information captured by the BDDs, we were able to determine how the sizes of the BDDs were affected by with the various parameters. The number of nodes in these particular state set BDDs grows as $O\left(rs(r+s)\log r + wo^2(r+s) + wo(r+s)^2\right)$ (this growth rate was obtained using the same type of analysis as that above). The largest BDDs we encountered had slightly less than 12,500 nodes; typical sizes were about 1,000 nodes.

We performed the tests described above using a CTL model checker written mostly in the T dialect of LISP [33]. The actual BDD manipulation routines are written in C and are roughly comparable to the package described by Brace, Rudell and Bryant [4]. The model checker was run on a SPARCstation 1+. Figure 5 shows how the verification time depends on the parameters r, w, s and o. This plot (and the other plots in this paper) uses a log scale on both axes. On such a plot, the polynomial relationship $y = x^n$ appears as a straight line with slope n. The following table shows the values used for the fixed parameters in these tests.

	r	w	s	o
vary r		1	1	1
vary w	4		1	1
vary s	2	2		1
vary o	2	4	1	

The verification time is dominated by the time required to compute the state sets for the subformulas $\mathbf{AX}^{s+2} reg_{i,j}$. There are rw such formulas. Each computation of this form involves one call to *RelProd* for each piece of the transition relation. The verification times can be accounted for as follows.

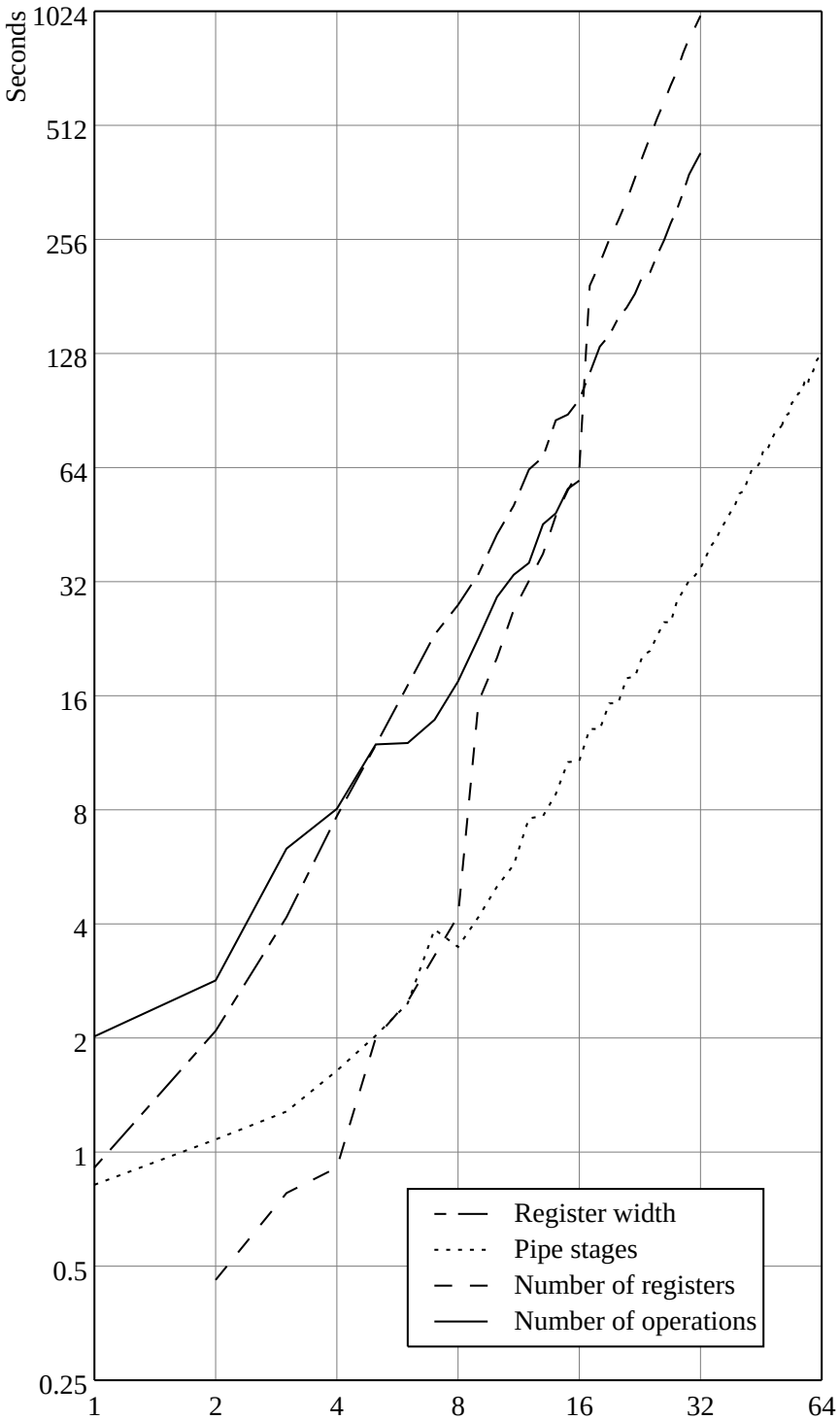

Figure 5: Pipeline circuit verification times

1. As r increases from $2^i + 1$ to 2^{i+1} for some i, the number of $\mathbf{AX}^{s+2}$ computations increases linearly. The number of pieces in the transition relation also increases linearly, and in each call to *RelProd*, the size of the result BDD increases linearly. If we make the assumption that the time to do a BDD operation is linear in the size of the result BDD, then we would expect these three linear increases to produce cubic growth in the verification time. The slopes of the best fit lines for r equal to 9 through 16 and for r equal to 17 through 32 are both 2.5. In the general case where r ranges over more than a factor of 2, we would expect the time to grow as $O(r^3 \log r)$, but we do not have enough data to completely substantiate this conjecture.

2. As w increases, the number of $\mathbf{AX}^{s+2}$ computations increases linearly and the size of the BDD resulting from each operation increases linearly. This leads us to expect quadratic growth in the verification time. The slope of the best fit line for w equal to 17 through 32 is 2.1.

3. As s increases, the number of $\mathbf{AX}$ computations needed to evaluate each formula of the form

   $$\mathbf{AX}^{s+1} reg_{j,i}$$

   increases linearly and the sizes of the BDDs produced within these steps increase linearly. When computing

   $$\mathbf{AX}^{s+2} reg_{j,i} = \mathbf{AXAX}^{s+1} reg_{j,i},$$

   the BDDs during the last $\mathbf{AX}$ operation grow quadratically. Overall we expect quadratic growth in the verification time. The slope of the best fit line for s equal to 33 through 64 is 1.8.

4. How the verification time varies with o depends on the particular operations used. The number of $\mathbf{AX}^{s+2}$ computations does not change. The BDDs for the state sets grow $O(o^2)$ for the operations mentioned above. We would thus expect quadratic growth in the verification time. The slope of the best fit line for o equal to 9 through 16 is 1.7.

It is important to note that in all cases, the verification time is growing polynomially in the number of *components* of these example circuits. Polynomial verification times were also documented in earlier work [12, 13, 14]. Other researchers [1] using symbolic techniques have demonstrated verification times that grow sublinearly in the number of *states* of the system, but still exponentially in the number of components.

For comparison, we also ran the verification with a monolithic transition relation. With 8 bit registers, the monolithic transition relation required more than 75,000 BDD nodes to represent, compared with fewer than 750 nodes using a partitioned transition relation, a difference of more than two orders of magnitude. In addition, the verification needed nearly an order of magnitude more time. We also note that combining parts of a transition relation can result in higher asymptotic complexity. For example, the total number of nodes in the BDDs that represent the register file in the partitioned transition relation is $O(r \log r)$, while the BDD for their conjunction has $O(r^2 \log r)$ nodes.

Partitioned transition relations can also be used to verify larger pipelines than those above. We verified a 32-bit wide pipeline with 8 general registers, 2 pipe registers, and one operation. This example had 406 state variables resulting in more than 10^{120} reachable states, and the verification took 1 hour and 25 minutes of CPU time on a SPARCstation 1+.

7.2 Other Synchronous Examples

This section gives empirical results for computing the set of reachable states of the MINMAX and KEY benchmarks.

The circuits of the MINMAX benchmark each consist of three control inputs and a data path of parameterizable width w. The data path is made up of a w bit input and three w bit state registers. The variable ordering we used is quite standard: control at top, data path variables interleaved and ordered most-significant bit to least-significant bit. We considered two different partitionings of the transition relation. The first had one BDD per bit of state, resulting in $3w$ BDDs each of size $O(w)$. With the ordering used, there was essentially no sharing of nodes between these BDDs, so the total number of nodes was $O(w^2)$. The second partitioning recombined the bits of each register (see section 5.4), resulting in 3 BDDs each still of size $O(w)$. Recombining the partitions reduced the total number of BDD nodes needed for the transition relation from $O(w^2)$ to $O(w)$.

The CPU time needed to compute the reachable states with the two representations shows a similar pattern (see figure 6). The graph shows CPU times in *tenths* of seconds on a Sun 3/60. The asymptotic complexity in the fully partitioned case grows slightly faster than quadratically, while the complexity with recombining is roughly linear. This compares well with the CPU time required by Berthet, Coudert and Madre [1], which grew exponentially with w. We also tried a least-significant bit to most-significant bit ordering. This reduced the total number of nodes in the transition relation with $3w$ partitions from $O(w^2)$ to $O(w)$, due to sharing. However, this did not affect the time required to compute the reachable states.

We also considered the KEY benchmark circuit[1]. The KEY benchmark circuit has 258 inputs (*start*, *encrypt* and key_0 through key_{255}), 228 state variables ($count_0$ through $count_3$, C_0 through C_{111} and D_0 through D_{111}) and 193 outputs. The transition functions for each of the $count_j$ state variables depend on *start*, *encrypt* and $count_i$ for $i \leq j$. The transition functions for each of the C_j depend on *start*, *encrypt*, C_j, D_j, $count_0$ through $count_3$, and two of the key_i inputs. The same is true of the transition functions for each D_j. Thus, the transition functions for each of the C_j and D_j depend on (have a *support* of) exactly 10 variables.

Because the size of the support of each transition function is small, the corresponding BDDs can be easily constructed for just about any variable ordering. Also, the particular supports for each state variable show that the KEY circuit can be naturally viewed as 113 communicating finite automata: one automata for the $count_0$ through $count_3$ state variables, and, for each j from 0 to 111, one automata containing the C_j and D_j variables. Each of these automata depend on the $count_j$ variables, so it is natural to put those variables at

[1]There are actually two sequential benchmark circuits called KEY, one with 228 latches [34] and one with 56 latches [19]. We use the one with 228 latches.

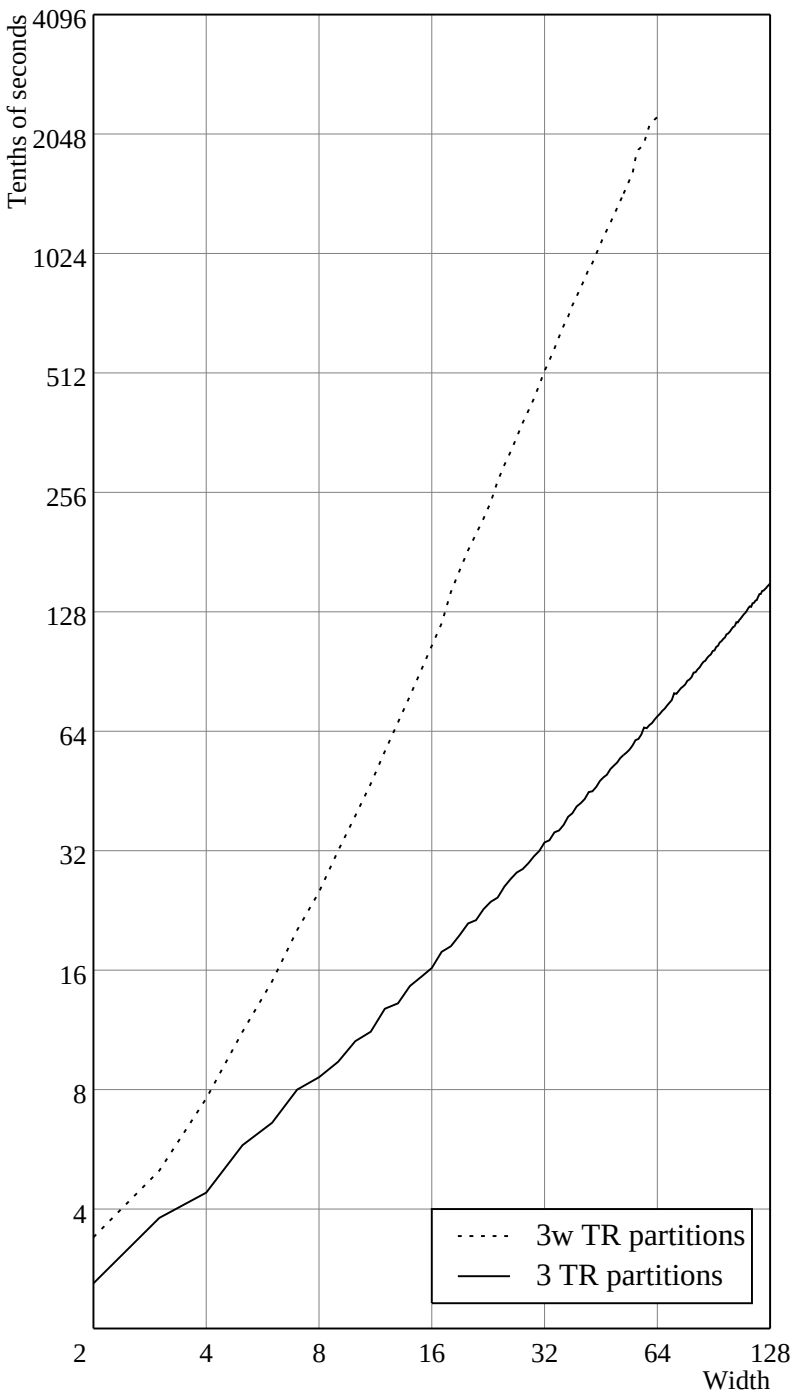

Figure 6: Verification times for MINMAX circuit

the top of the variable ordering. Also, the C_j and D_j should be interleaved; we used the ordering $C_{111}, D_{111}, \ldots, C_0, D_0$ in our experiments. With this ordering of the state variables, the largest state set BDD has 5584 nodes, which is an average of less than 25 BDD nodes per state variable. This small size is a result of limited communication between the 113 automata described above.

If full partitioning is used, the time necessary to compute the reachable states does not depend critically on the ordering of the input variables. However, the ordering can be important if parts of the transition relation are recombined. We put the *encrypt* and *start* inputs at the top of the ordering, and placed the key_i inputs near the C_j and D_j that depended on them. This ordering made it possible to use three partitions: one for the $count_j$ variables, one for the C_j variables and one for the D_j variables. The BDDs for the partitions had 33, 2464 and 2566 nodes, respectively. The time required to find the reachable states of the KEY benchmark circuit was 1019 seconds (CPU time on a SPARCstation 1+) when using a fully partitioned transition relation and 41 seconds for the three partition case, a speed up of nearly a factor of 25.

8 Verifying Asynchronous Circuits

One aspect of verifying speed-independent asynchronous circuits is checking that the circuit has no *hazards*. A hazard is informally defined as a state in which a gate can transition, and in which another transition can disable the output transition of the gate. This definition of hazards covers both static and dynamic hazards. In a real circuit, a hazard may result in the output of the gate starting to change and then returning to its previous state, with the result that parts of the circuit may see the transition and others may not. We can check for hazards in two steps. First, we compute the set of states that the circuit and its environment can reach from a given set of initial states. Then we check that none of these states results in a hazard. Finding the reachable states is the most computationally expensive of these two steps. In practice, checking for hazards is usually done as the reachable states are computed. This method can be generalized to handle a wide variety of safety properties of asynchronous circuits [22]. The set of reachable states is computed using the methods described in section 4.

8.1 Modified Breadth First Search

Recall that asynchronous circuits can be modeled using either conjunctive or disjunctive partitioned transition relations. These correspond to non-interleaving and interleaving semantics, respectively. There are significant differences in the complexity of doing reachability analysis using the two models. Consider two uncoupled systems M' and M'' with disjoint sets of state variables V' and V''. Let M be the composition of these two systems, and let $V = V' \cup V''$. This is an unrealistic example, but it helps illustrate what happens when computing the reachable states of loosely coupled systems. The BDD $S(V)$ representing the set of reachable states of M is equal to $S'(V') \wedge S''(V'')$, where $S'(V')$ is the BDD representing the reachable states of M' and $S''(V'')$ is the BDD representing the reachable states of M''. For simplicity, assume that the sets $S(V)$, $S'(V')$ and $S''(V'')$ are independent of whether

interleaving or non-interleaving semantics are used. An efficient way to order the BDD variables of the combined system in this case is to have all the variables of one component (say M') precede all of the variables in the other component. Then the number of BDD nodes in $S(V)$ is equal to the sum of the nodes in $S'(V')$ and $S''(V'')$. However, in spite of our assumption that interleaving and non-interleaving semantics give the same reachable states, the sizes of the BDDs representing the intermediate state sets are potentially different for the two semantics.

Let $S_i(V)$, $S'_i(V')$ and $S''_i(V'')$ be the BDDs representing the states reachable in i or fewer steps by M, M' and M'', respectively, using non-interleaving semantics. Similarly, let $T_i(V)$, $T'_i(V')$ and $T''_i(V'')$ be the BDDs representing the states reachable in i or fewer steps by M, M' and M'', respectively, using interleaving semantics. In the conjunctive (non-interleaving) case, $S_i(V) = S'_i(V') \wedge S''_i(V'')$, so the size of each $S_i(V)$ is equal to the sum of the sizes of $S'_i(V')$ and $S''_i(V'')$, just as for the set of reachable states. For the disjunctive case, if a global state is reachable in at most i steps and the local state of M' is reachable in k steps, then the local state of M'' must be reachable in at most $i-k$ steps. Hence,

$$T_i(V) = \bigvee_{k=0}^{i} \left(T'_k(V') \wedge T''_{i-k}(V'') \right).$$

Thus, interleaving semantics introduces an artificial correlation between the local states of M' and M'' in the $T_i(V)$. In practice, the $T_i(V)$ are generally much larger than the $S_i(V)$. Because of this effect, standard breadth first reachability analysis with disjunctive partitioning is less efficient than with conjunctive partitioning.

We can make disjunctive partitioning more efficient by using a modified breadth first search (MBFS) for reachability analysis. To search the reachable states of M, first compute states reachable by transitions of wires in M'. Then compute the states reachable from that set by transitioning on wires in M''. This is equal to the global reachable state set, since M' and M'' are uncoupled. Separately computing local fixed points for the two parts of the system in this way removes the artificial correlation described above.

In general, for a circuit C divided into loosely coupled subcircuits C_j, we compute the reachable states of C by repeatedly computing local fixed points for each C_j until a global fixed point is reached. This idea can be extended to a hierarchy with any number of levels. For example, consider a closed system with 4 subcircuits C_0 through C_3 (see figure 7). Let V_i be the set of state variables of C_i, and let V be the union of the V_i. Subcircuits C_i and C_k communicate through the state variables in $V_i \cap V_k$. Let O_i be the set of variables in V_i that are driven by C_i. The O_i are pairwise disjoint and their union is equal to V. We can construct a hierarchy where the top level splits the circuit into 2 parts: C_0 together with C_1 form one part, C_2 and C_3 form the other. The second level of the hierarchy further splits the circuit into the individual C_i. In this case modified breadth first search proceeds by alternately finding all the states reachable via transitions in $O_0 \cup O_1$ and in $O_2 \cup O_3$ until a fixed point is reached. At each iteration, finding the states reachable via $O_0 \cup O_1$ is done by alternately finding all the states reachable via transitions in O_0 and in O_1 until a fixed pointed is reached.

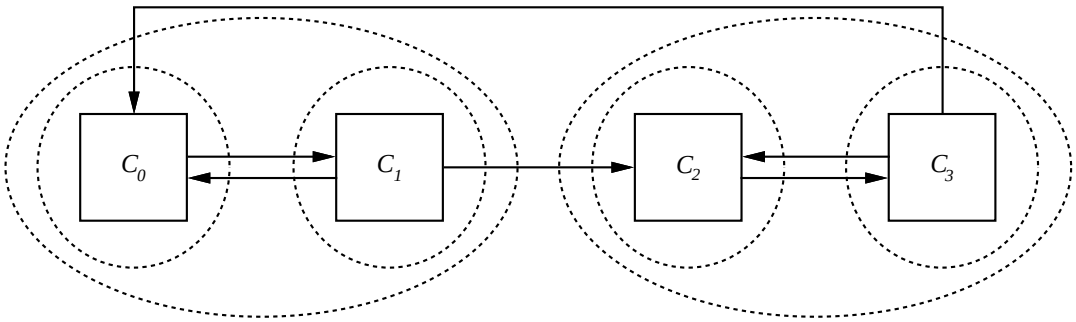

Figure 7: Example hierarchy for modified breadth search

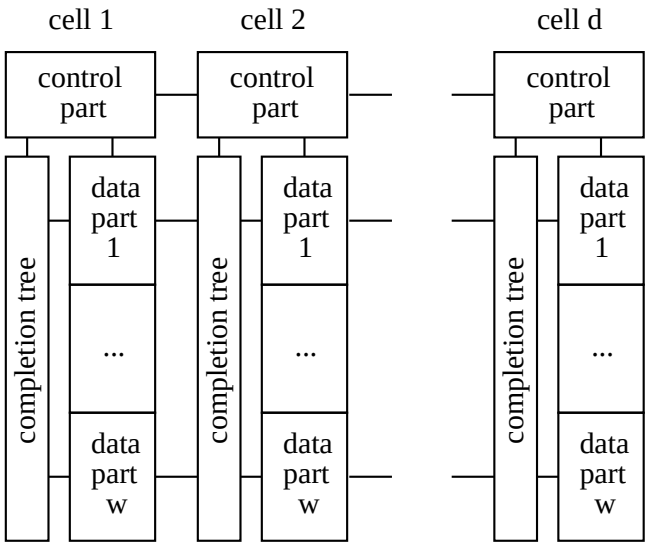

Figure 8: Stack circuit block diagram

8.2 An Asynchronous Stack

In this subsection, we compare conjunctive and disjunctive partitioned transition relations for verifying asynchronous circuits by considering an asynchronous lazy stack due to Martin [28]. To determine the asymptotic performance of the various methods discussed above, we performed a reachability analysis for stacks with varying depth d and word width w. This is sufficient to study the asymptotic complexity of verification, even though we did not check for hazards. Hazard checking increases the verification times by about a factor of two.

Figure 8 shows a block diagram of the stack. It consists of an array of d cells, each cell consisting of a control part, a data part and a completion tree. The data part of each cell consists of w storage elements. A completion tree consists of $(w-1)$ C-elements, each with 2 inputs. It effectively acts as a w-input C-element and is used to signal when all the storage elements in a cell have completed the current data transfer. The model that we used also included an environment for the stack that nondeterministically performs push and pop operations.

The variable ordering that we used can be understood in relation to figure 8. We ordered

the variables by scanning the figure from top to bottom, and, within each row, by scanning from right to left. Thus, we had variables for the control part of cell d first, the control part of cell $d-1$ next, etc. After all of the control parts, we had data part 1 for cells d through 1, together with the completion tree variables derived from those data parts. The last variables in the order were those for data part w in cell 1 (and the associated completion tree variables).

We did a detailed study of how verification time varied with w for three different methods:

1. Disjunctive partitioning using modified breadth first search. We combined the transition relations for the gates making up each individual control part, each of the individual storage elements, and each completion tree. At the top level, the hierarchy used for local fixed point computation consisted of the environment and each cell as a unit. Each cell was broken into the control part, the completion tree and the data part. The data part was further subdivided into a hierarchy consisting of a balanced binary tree with $\lceil \lg(w) \rceil$ levels.

2. Conjunctive partitioning using the same partitioning of the transition relation as above. We used the following ordering ρ of the parts of transition relation:

   (a) the environment at the top of the stack;

   (b) the control part and data parts for cell 1, followed by the control and data parts for cell 2, etc.

   (c) the completion tree for cell 1, followed by the completion tree for cell 2, etc. and

   (d) the environment at the bottom of the stack.

3. Conjunctive partitioning using the same partitioning as above, but with the control and data parts within each cell combined into one BDD. The ρ used above is modified in the obvious way.

In all cases, we used an initial state set in which each cell could be full or empty and the data in each cell was arbitrary. Using a more restricted set of initial states, such as having all cells initially empty, can increase the verification time by as much as a factor of d. Interleaving semantics (method 1) and non-interleaving semantics (methods 2 and 3) both produced exactly the same set of reachable states for the stack circuit.

We also experimented with disjunctive partitioning using standard breadth first search. However, we found that this method was feasible only for small examples. Disjunctive partitioning with modified breadth first search and conjunctive partitioning were both much more efficient.

A graph of the search times versus stack width for the three methods is shown in figure 9. Search times using methods 1 and 2 grew at about $w^{2.2}$ and $w^{1.8}$, respectively. Method 3 gave a growth rate of roughly $w^{1.2}$. Using this method, we were able to find the reachable states of a 32 bit wide, depth 2 stack in 38 minutes of CPU time on a SPARCstation 1+. This circuit had 989 boolean state variables and over 10^{50} reachable states.

The BDDs in the transition relation are all of constant or linear size, except for those representing the completion trees. For both interleaving and non-interleaving semantics, a

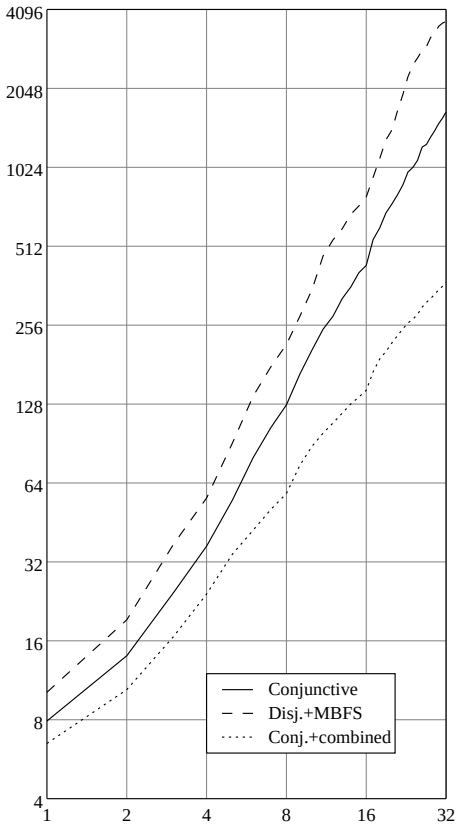

Figure 9: Search times in seconds for stacks of various widths, with $d = 1$

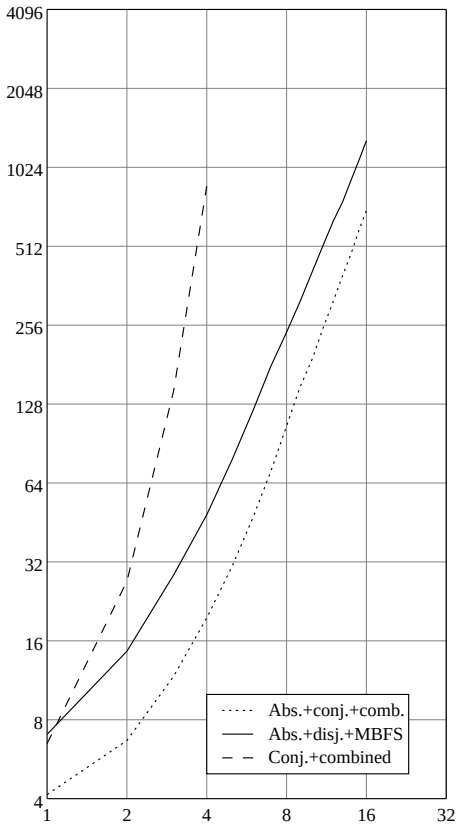

Figure 10: Search times in seconds for stacks of various depths, with $w = 1$

simple analysis of the BDDs for the completion trees show that for w equal to a power of 2, the number of nodes $f(w)$ must satisfy the equation (to first order)

$$f(2w) = 3f(w).$$

When f is extended to values of w that are not powers of 2, it is still a monotonically increasing function. As a result, the above equation is sufficient to show that $f(w)$ is $O(w^{\lg 3})$. For the values of w we considered in our experiments, a single BDD for each completion tree requires only a small number of nodes. However, for larger w, it might be necessary to split the completion trees into more than one BDD.

We also explored how the search time varied with the depth of the stack (see figure 10). The number of steps needed to compute the reachable states grows quadratically in d. The states which require the largest number of steps to reach are states in which internal signals within the stack control are not stable. Thus, we were able to avoid the quadratic search depth by replacing the control part of each cell by an abstract model having only external signals. We separately verified (using a variant of Dill's explicit state verifier [22]) that the abstract model correctly describes the external behavior of the control part. With this abstraction, the number of steps needed to find a fixed point is linear in d. The search times grow as $d^{2.4}$ for disjunctive partitioning and as $d^{2.7}$ for conjunctive partitioning.

Although this kind of abstraction can greatly improve the efficiency of verifiers that explicitly enumerate states, it is usually not nearly as helpful when used with symbolic verifiers. For example, the search times for stacks of depth one improve only about 20 percent when the abstract model of the control part is used. The effect abstraction on the search depth is an exception to this rule.

The maximum size of the state set BDDs encountered during the searches are shown in figures 11 and 12. The state sets grow slightly faster than linearly with width (probably due to the completion trees). They grow approximately quadratically with depth when we use the abstract model of the control part of each cell.

8.3 Distributed Mutual Exclusion

As another example, we considered the verification of an asynchronous circuit for ensuring mutually exclusive access to a shared resource. This circuit is also due to Martin [27, 23]. The circuit consists of a ring of c cells. Each cell communicates with a user of the resource and with its left and right neighbors in the ring. Mutual exclusion is ensured by having a single "token" that is passed around the ring. A cell must have the token before granting access to its user. The distributed mutual exclusion circuit is an example of an asynchronous circuit with complex control and no data path.

The variable ordering we used had the variables for each cell grouped together. The first variables in the order were those for cell 1, and the last were those for cell c.

We studied how the complexity of reachability analysis varied with c for a variety of cell models and search techniques:

1. Disjunctive partitioning using modified breadth first search. We combined the transition relations for the gates making up each individual cell, so the number of elements in the partitioning was equal to the number of cells. The hierarchy used for local fixed

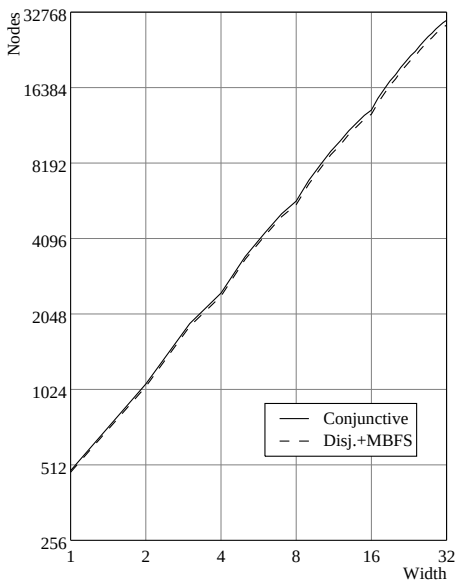

Figure 11: State set BDD sizes for stacks of various widths, with $d = 1$

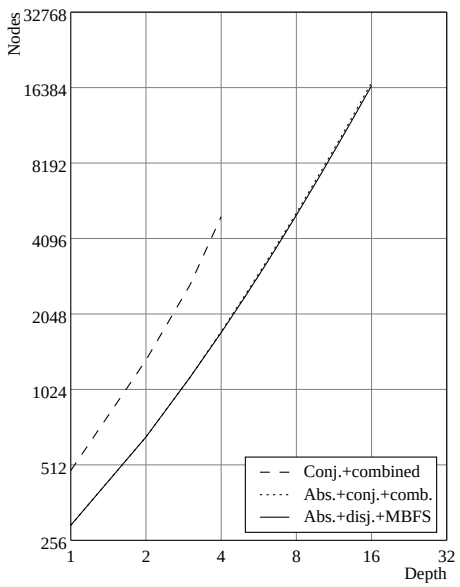

Figure 12: State set BDD sizes for stacks of various depths, with $w = 1$

point computation was obtained by repeatedly splitting the set of cells in half. The cells that were connected in the circuit were grouped together in the hierarchy.

2. Conjunctive partitioning with the first $c-1$ cells combined and the last cell as a separate part of the transition relation. We left the last cell separate because it introduces constraints between some of the variables at the top and bottom of the BDD variable ordering. The last cell was processed first, followed by the combined group of $c-1$ cells.

3. Disjunctive partitioning as in item 1, but using an abstract model of the cell.

4. Conjunctive partitioning as in item 2, but using an abstract model of the cell.

In all cases, we used an initial state set of c states, each with the token in a different cell. Interleaving semantics (methods 1 and 3) and non-interleaving semantics (methods 2 and 4) both produced exactly the same set of reachable states for the distributed mutual exclusion circuit.

A graph of the search times versus number of cells for the various methods is shown in figure 13. Disjunctive partitioning with modified breadth first search and conjunctive partitioning were again roughly comparable, with the former having a lower asymptotic complexity. This contrasts with the stack example, where the combined conjunctive partitioning was faster. This difference is probably because the complexity of the stack is in its data path, while the complexity of the mutual exclusion circuit is due to control rather than data. The verification times for the four methods grow as c to the power of 2.1, 3.1, 2.3 and 3.5, respectively. The largest unabstracted circuit that we examined had 16 cells, 256 boolean state variables, and over 10^{16} reachable states. It took slightly less than 30 minutes of CPU time on a SPARCstation 1+ to find the reachable states.

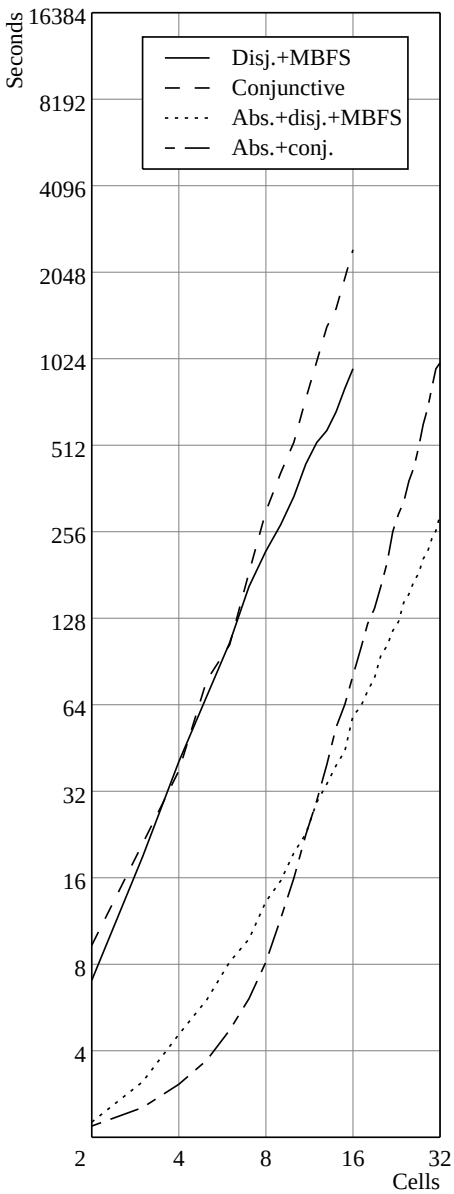

Figure 13: DME circuit verification times

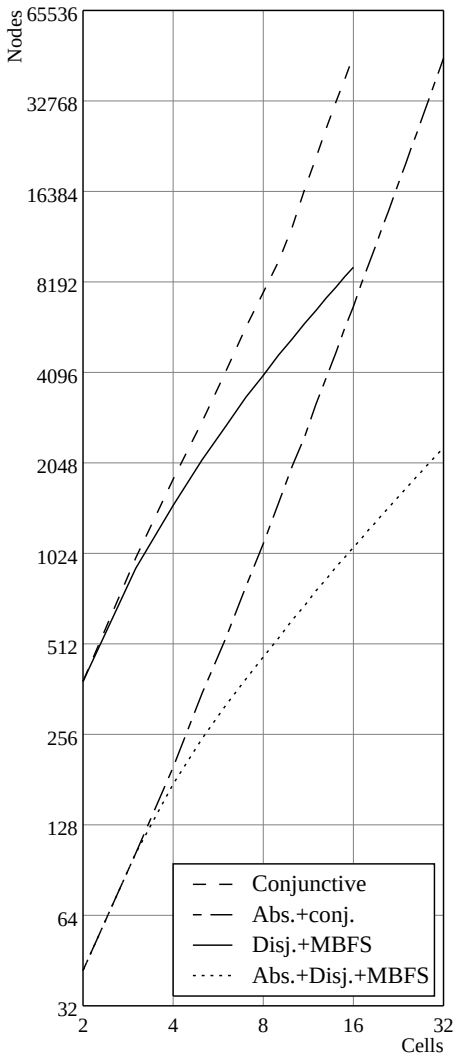

Figure 14: State set BDD sizes for DME circuit

The total number of BDD nodes needed to represent the transition relation grew linearly in c in all cases. The maximum state set BDD sizes are shown in figure 14. In the case of the conjunctive methods, these BDDs grow approximately cubicly, while with disjunctive partitioning and MBFS, the growth rate is reduced to linear.

9 Discussion

We have described a BDD-based algorithm for CTL model checking with fairness constraints. The use of modified breadth first search for reachability analysis has also been described, as well as the advantages of viewing reachability analysis as a method for constructing and checking invariants. All of these methods are made significantly more efficient by the use of partitioned transition relations. We have empirically studied the asymptotic complexity of verifying both synchronous and asynchronous circuits. In all cases, the verification time for

these circuits grew as a small polynomial in the number of circuit components.

Two of the distinguishing features of our verification methods are the use of transition relations and the amount of guidance the user provides to the verifier. These features are discussed in more detail below.

9.1 Transition Relations

Our verification methods use relations to describe how circuits can transition from one state to another. We considered both monolithic transition relations (which are represented by a single BDD) and partitioned transition relations (more than one BDD). For deterministic systems, this information can be represented using transition function vectors instead. In this method, a separate BDD is used for each Boolean state variable of the system. This BDD represents the function computed by the combinational logic driving the associated latch. Coudert *et al.* [18, 20] describe a number of algorithms for manipulating transition functions.

Of these three methods of representing transitions (transition functions and monolithic and partitioned transition relations), we believe a partitioned transition relation usually gives the best performance. A monolithic transition relation can require many more BDD nodes than a corresponding transition function vector [20] or partitioned transition relation. However, when a monolithic transition relation is not too large to store in semiconductor memory, computations with the transition relation appear to be faster than those using transition functions. This observation was also made by Coudert *et al.* [20] when they compared transition relations to their techniques based on transition functions. In addition, while we have demonstrated polynomial growth in verification time for several classes of circuits using transition relations, no state exploration method based on transition functions has shown such results. Our empirical results indicate that partitioned transition relations give both the speed of transition relations and the memory efficiency of transition functions.

Touati *et al.* [34] independently proposed another method for representing transition relations as implicit conjunctions. They use the *constrain* operator of Coudert *et al.* [18] to eliminate the state set $S(V)$ in equation 2. Then they compute the resulting conjunction as a balanced binary tree, quantifying out each variable in V when all the BDDs depending on that variable have been combined. We believe that this method is inferior to the one proposed here because the constrain operator may introduce dependencies on any of the variables in $S(V)$. Generally, $S(V)$ depends on all or nearly all of the variables in V. Thus, after applying the constrain operator, all of the partitions of the transition relations may depend on most of the variables in V. As a result, it may not be possible to quantify out many variables before performing the final conjunction, greatly reducing the effectiveness of early quantification. Touati *et al.* also suggest having one transition relation per state variable. As we have described, it is often better to combine parts of the transition relations. This idea is also applicable to their method.

We implemented their method and tested it on some of the examples in section 7. For a pipeline with four 8 bit registers and one pipe register, our method was more than five times faster. In addition, for some of the relational product computations, the intermediate BDDs using their method were more than an order of magnitude larger than the final result.

The two methods have been applied to the KEY benchmark and the MINMAX benchmark with a 32 bit wide data path. We computed the reachable states of these circuits in 41 seconds (CPU time on a SPARCstation 1+) and less than 4 seconds (CPU time on a Sun 3/60), respectively (see section 7.2). Touati *et al.* [34] reported run times (on a DEC 5400) of 5706 seconds and 444 seconds, respectively. Their data includes the time needed for parsing input files, computing the reachable states of the product automata of two identical circuits, and checking equality of the outputs of the two automata. Although these times are difficult to compare directly, a speed up of two orders of magnitude suggests that our method performs better on these two benchmarks. Empirical results on additional benchmarks are required, however, before a definitive conclusion can be reached.

9.2 Degree of Automation

State exploration based verification methods tend to be more automatic then methods based on theorem provers. This is particularly true when attempting to verify a circuit that is not correct. State exploration methods can easily produce a counter-example trace that helps the user find errors in the circuit. With a theorem prover, the user only knows that the attempted proof will not go through, without knowing whether this is because of a circuit error or a weakness in the theorem prover.

Although symbolic state exploration methods are much more automatic than using a theorem prover, it is still necessary for the user to do more than just provide a specification and a circuit description. In this section, we consider some of the decisions that the user must make.

First, the user must choose one of the many techniques in the literature, such as forward and reverse reachability analysis, CTL model checking, etc. This is a difficult decision; determining general rules about which methods perform well on what kinds of circuits is still an open research question. There may be no alternative other than to try several methods. It is useful to start with a small model of the circuit to be verified, either by abstracting out much of the functionality of the circuit, or as we have done in this paper, by parameterizing the data path width, number of registers, etc. If a particular method performs better on a smaller version of the circuit, it is likely to perform better on the full circuit, as well.

Example verification attempts on particular circuits, such as those in this paper, are also helpful. We described three different methods: CTL model checking, and forward and reverse reachability analysis. Only model checking worked well on the synchronous pipelined ALU circuit that we considered. With forward reachability analysis the BDD needed to represent the set of states reachable in one step was exponential in the size of the circuit. However, when we checked for hazards in two asynchronous circuits, forward reachability analysis was quite efficient. In this case, the set of states with hazards was represented by an implicit disjunction of BDDs, one BDD for each component of the circuit. If we had used reverse reachability analysis or CTL model checking, then we would have had to do a separate analysis for each of the disjuncts. Also, the BDDs for reverse reachability analysis of asynchronous circuits tend to be much larger than for forward analysis; the structure inherent in the set of reachable states is lost.

With all BDD-based verification methods, a good variable ordering must be found. When

partitioned transition relations are used, the most important factor is the size of the BDDs representing the many state sets computed during verification. In our experiments, we often found ways to improve a variable ordering by trying it on a small version of the circuit. We plotted *profiles* of the BDDs that were computed. A profile is a graph that shows, for each variable, the number of BDD nodes labeled by that variable. Profiles can provide information about how information flow in the circuit results in large BDDs, and how the variable order might be modified to make the BDDs smaller. We found that a little time pondering variable orders paid off with drastically reduced verification times.

The user must also provide a partitioning of the transition relations. This is not as critical as the variable ordering; reasonable results can be obtained by simply having one partition for each state variable of the circuit. However, we have shown that even better results can be obtained by combining some of the partitions together. Finding a good way to combine partitions was not a problem; usually our first guess worked quite well. We suspect the process could be easily automated, given information about the hierarchical structure of the circuit.

It appears more difficult to choose automatically the order in which partitions are used when computing relational products. Nonetheless, in practice it was not difficult to choose an ordering by hand. The orderings used in our experiments were based on the natural flow of information in the circuits. Again, we found it helpful to test our choices on a small version of the circuit being verified.

If modified breadth first search is used, then partitioning the transition relation is made slightly more complicated by the inclusion of hierarchical information to guide the search. Assume the user has already chosen a partitioning for standard breadth first search. Then, given an explicit representation of the hierarchical structure of the circuit, it is straightforward to automate the process of finding a hierarchy to guide a modified breadth first search.

Viewing reachability analysis as a way of helping the user construct an invariant provides a way for the user to help the machine produce the invariant by changing the set of initial states. For the asynchronous circuits we considered, it is easy to manually produce an expression for all of the reachable quiescent states of the circuit. We suspect this is true in general. The verifier performed quite well on our example circuits when the set of quiescent states was used as a starting point for constructing the invariant.

In our experience, all of the decisions the user must make to use our verification methods are straightforward given an understanding of the circuit's operation and of the basic properties of BDDs. Some of these decisions could be easily automated; other areas appear better left to the user, at least given the current state of the art. Although providing these hints to the verifier requires some extra effort on the part of the user, it is often justified by the significant improvement in performance that can result.

What is the best balance between automation and manual effort in a BDD-based verification method? The answer to this question depends on the situation in which the method is to be used. If the goal is to produce a verification tool that can be used with a minimum of training and without expert assistance, then full automation is of paramount importance. The power of the methods described here could not be used fully in such a verification tool. Other more automatic methods [1, 24, 34] might be more appropriate in this situation.

Although fully automatic verification methods have become much more powerful in the last several years, there are still severe restrictions on the size of the circuits to which they can be applied. Our empirical results suggest that a small amount of manual assistance can greatly improve the scalability of BDD-based verification techniques. Improving scalability requires more than just a constant factor speed up; it requires a drastic reduction in the rate that verification time increases as a function of increasing circuit size (for example, exponential growth reduced to quadratic growth). Such a reduction in growth rate can only be demonstrated by asymptotic analysis, such as the kind of empirical analysis used in this paper.

If further research confirms that manual assistance can improve scalability, then we see two ways that development of manually assisted verification methods can have direct practical value. The first, quite naturally, is applying these methods to verification problems that are beyond the state of the art for fully automatic verification tools. Manual assistance would still be potentially costly, in terms of time and necessary expertise, but formal verification would not be possible otherwise in this situation. The second use would be to shed light on potential improvements to existing fully automatic techniques. We view the current paper as an example of this. We have described the kind of manual assistance required in our methods; if these parts of the verification process could be efficiently automated, the result would be a more powerful fully automatic verification technique.

References

[1] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE International Conference on Computer Design*, 1990.

[2] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, October 1989.

[3] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, pages 759–764, November 1989.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[5] M. C. Browne. An improved algorithm for automatic verification of finite state machines using temporal logic. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, Boston, Mass., June 1986.

[6] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

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

[8] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, 1991.

[9] R. E. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In R. Kurshan and E. M. Clarke, editors, *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1990. Also in Springer-Verlag LNCS 531.

[10] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, 1991.

[11] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[14] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[15] H. Cho, G. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *IEEE International Conference on Computer-Aided Design*, pages 134–137, 1990.

[16] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, 1981. Springer-Verlag.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[18] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[19] O. Coudert and J. C. Madre. A unified framework for the formal verification of circuits. In *IEEE International Conference on Computer-Aided Design*, pages 126–129, 1990.

[20] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In R. Kurshan and E. M. Clarke, editors, *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1990. Also in Springer-Verlag LNCS 531.

[21] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*. MIT Press, 1988.

[22] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988. Also appeared as [23].

[23] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[24] T. Filkorn. A method for symbolic verification of synchronous circuits. In *Proceedings of the Tenth International Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[25] R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121–861010–33–TM, Bell Laboratories, 1986.

[26] B. Lin, H. J. Touati, and A. R. Newton. Don't care minimization of multi-level sequential logic networks. In *IEEE International Conference on Computer-Aided Design*, pages 414–417, 1990.

[27] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *Chapel Hill Conference on VLSI*, 1985.

[28] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *IEEE International Conference on Computer Design*, October 1987.

[29] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[30] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. To appear.

[31] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessors*, 1991.

[32] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. M. Clarke, editors, *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1990. Also in Springer-Verlag LNCS 531.

[33] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual*. Yale University, 4th edition, 1984.

[34] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *IEEE International Conference on Computer-Aided Design*, pages 130–133, 1990.