Sequential Circuit Verification Using Symbolic Model Checking

J. R. Burch E. M. Clarke K. L. McMillan
Carnegie Mellon University

David L. Dill
Stanford University

Abstract

The temporal logic model checking algorithm of Clarke, Emerson, and Sistla [10] is modified to represent state graphs using *binary decision diagrams* (BDDs) [6]. 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^{20} states. Our model checking algorithm handles full CTL with fairness constraints. Consequently, we are able to handle 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 the realization of the circuit in hardware. This fact has stimulated interest in formal verification techniques for hardware designs. A number of different techniques have been proposed, but nearly all

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 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. The third author is supported by an AT&T Bell Laboratories Ph.D. Scholarship. The fourth author is 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 or the U.S. government.

can be classified in terms of the natural division between the *data paths* and the *controlling circuitry* in digital devices. The most successful methods to date for verifying data path logic treat only functional behavior, without considering sequential behavior [17]. 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. The most effective techniques for reasoning about sequential behavior, on the other hand, usually require a complete exploration of the state space of the circuit [5, 12, 14]. The state enumeration 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 excessively large. The approach has not been very useful with data path circuits, 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, however, 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 a technique for reasoning about sequential circuits, called *temporal logic model checking* [9, 10], can modified to represent state graphs using *binary decision diagrams* (BDDs) [6]. Because this representation captures some of the regularity in the state space determined by the data path logic, we are able to verify sequential circuits with an extremely large number of states. The algorithm is based on computing fixed points of functions, called predicate transformers, which map sets of states to sets of states. Predicate transformers are used to model the way circuits transition from one state to the next. 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 algorithm on both synchronous and asynchronous circuits with data path logic.

Previously, most of the applications of BDDs have been to the verification of combinational circuits. However, there have been some recent applications to sequential circuits. One approach uses a symbolic switch-level simulator, in which a sequence of operations is simulated with symbolic inputs. The use of symbolic inputs allows one to verify that certain pre- and post-conditions are satisfied independently of the actual input values applied. This technique has been used by Bryant to verify a MOS memory circuit [7]. A second approach due to Bose and Fisher [2] verifies a pipeline circuit with respect to a simpler abstract model by means of a representation function, in analogy to abstract data type verification.

While both of these approaches are quite powerful for reasoning about certain classes of circuits, they clearly require much more effort from the user than state enumeration methods. In each of these approaches, the user must give a step-by-step specification using pre-condition, post-condition notation, instead of describing the behavior over time with a single temporal formula. The method of Bose and Fisher also requires that the user provide the analog of a data type invariant. An even more serious drawback stems from the limited expressive power of ordinary propositional logic for this type of application. Since they are unable to express unbounded execution histories in propositional logic, their techniques cannot be easily extended to systems of controllers that operate concurrently, nor can they deal with *liveness* properties, which state that an event must occur at some point in the future but do not provide an explicit time bound on when the event should occur.

Coudert, Berthet, and Madre describe a system for checking equivalence between deterministic finite automata [11]. Their system performs a breadth-first search of the state space determined by the product of the two automata. The set of reachable states is represented using a BDD, and in this sense, their method is closely related to our own. However, unlike the technique described in this paper, their method does not deal with indeterminate computations, asynchronous circuits or liveness properties.

Fujita and Fujisawa [13] describe a verification procedure based on linear temporal logic that uses binary decision diagrams to represent the transition conditions in automata derived from temporal logic formulas. However, their technique still suffers from a form of the state explosion problem, because they represent states explicitly in automata derived from temporal formulas. In our work, as in the work by Coudert, Berthet, and Madre, binary decision diagrams are used to represent both the transition relation of the model and subsets of the state space, so that the state graph is never explicitly constructed.

Bose and Fisher [3] have described a BDD-based algorithm for CTL model checking that is applicable to synchronous circuits. They do not provide empirical results on the algorithm's performance, however. In addition, their algorithm does not handle fairness constraints [10], so it is of limited use in proving liveness properties.

2 CTL and Model Checking

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL or Computation Tree Logic [10]. In this logic each of the usual forward-time operators of linear temporal logic (**G** *globally* or *invariantly*, **F** *sometime in the future*, **X** *nexttime* 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:

1. Every *atomic proposition* is a formula in CTL.
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 $\mathbf{EG}f$.

The semantics of a CTL formula is defined with respect to a *labeled state transition graph.* A labeled state transition graph is a 5-tuple $\mathcal{M} = (P, S, L, N, S_0)$ where P is a set of atomic propositions, S is a finite set of states, L is a function labeling each state with a set of atomic propositions, $N \subseteq S \times S$ is a transition relation, and S_0 is the set of initial states. Throughout this paper, for any set R, we say the predicate $R(a, b)$ is true if and only if $\langle a, b\rangle \in R$. This notation is used to define a *path* as an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $N(s_i, s_{i+1})$ is true 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. If a CTL formula f is an atomic proposition, then f is true of a state s if and only if $f \in L(s)$. **X** is the *nexttime* operator: $\mathbf{EX}f$ will be true in a state s of $\mathcal{M}$ if and only if s has some successor s' such that f is true at s'. **U** is the *until* operator: $\mathbf{E}[f \;\mathbf{U}\; g]$ will be true in a state s of $\mathcal{M}$ if and only if there exists a computation path starting in s and 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. The operator **G** is used to express the *invariance* of some property over time: $\mathbf{EG}f$ will be true at a state s if there is a path starting at s such that f holds at each state on the path. If f is true in state s of structure $\mathcal{M}$, we write $\mathcal{M}, s \models f$. We write $\mathcal{M} \models f$ if $\mathcal{M}, s \models f$ for all states s in S_0. We will identify a CTL formula f with the set $\{s : \mathcal{M}, s \models f\}$ 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 each *next* state).
- $\mathbf{EF}f \equiv \mathbf{E}[\mathit{true}\;\mathbf{U}\; f]$ 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 *globally* or *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).

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 [4, 10]. However, an explosion in the size of the model may occur when the labeled state transition graph is extracted from a circuit, particularly if the circuit contains many registers or other memory elements. The new model checking algorithm described in this paper was developed to help alleviate this problem.

3 Binary Decision Diagrams

Binary decision diagrams (BDD) are a canonical form representation for a boolean formulas. Bryant described algorithms for efficient manipulation of BDDs [6]. They are often substantially more compact than the traditional normal forms such as conjunctive normal form and disjunctive normal form, and hence have found application in symbolic verification of combinational logic, among other uses. 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 satisfies the formula by traversing the graph beginning at the root, branching at each node based on the assigned value of the variable which labels that 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 this assignment satisfies the formula.

Bryant showed that there is a unique BDD for a given formula with a given variable ordering. The size of a BDD depends critically on the variable ordering. Bryant also gave algorithms of low complexity for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for formulas f and g. The only other operations which we require for the algorithms

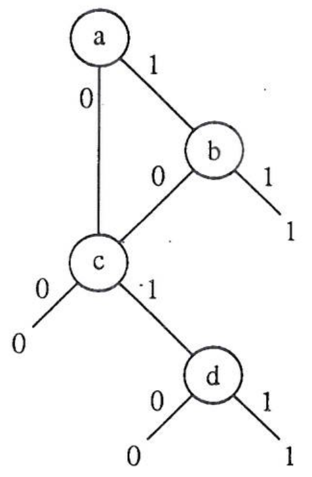

Figure 1: A Binary Decision Diagram

that follow are quantification over boolean variables and substitution of variable names. Bryant described an algorithm for computing the BDD for a restricted formula of the form $f|_{a=0}$ or $f|_{a=1}$. 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 $f|_{a=0} \vee f|_{a=1}$. The substitution of a formula g for a variable v in a formula f, denoted $f\langle v \leftarrow g\rangle$ can be accomplished using quantification, since

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

More efficient algorithms are possible, however, for the case of quantification over multiple variables, or multiple renamings.

4 Representing Transition Graphs

Let $\mathcal{M} = (P, S, L, N, S_0)$ be a labeled state transition graph. When representing $\mathcal{M}$ symbolically, we assume that any two states in $\mathcal{M}$ are uniquely identified by the truth values they assign to the atomic propositions:

Assumption 1 *For any two states s_1 and s_2 in S, if $L(s_1) = L(s_2)$, then $s_1 = s_2$.*

This assumption causes no loss of generality since extra atomic propositions can be added in order to make $L(s_1) \neq L(s_2)$ for any two distinct states s_1 and s_2. Let V_P be the set of possible truth assignments to the atomic propositions in P. We write $\bar{u}$, $\bar{v}$, $\bar{w}$, etc., to denote elements of V_P. Thus,

$\bar{v}(p) = 1$ if the atomic proposition p is true in the truth assignment $\bar{v}$; otherwise, $\bar{v}(p) = 0$. Any $\mathcal{M}$ satisfying the above assumption is isomorphic to a labeled state transition graph with a subset of V_P as its set of states. This isomorphism maps a state s to the $\bar{v} \in V_P$ such that

$$L(s) = \{p \in P : \bar{v}(p) = 1\}.$$

Thus, there is no loss of generality in

Assumption 2 *$\mathcal{M} = (P, S, L, N, S_0)$ where $S = V_P$ and*

$$L(\bar{v}) = \{p \in P : \bar{v}(p) = 1\}.$$

Satisfying this may require adding additional states to the transition graph, but these states will be unreachable and will not affect the truth of any CTL formulas. We need not represent the state set S or the labeling function L explicitly since they will always have the form described in assumption 2. Thus, for our purposes, a labeled state transition graph is completely determined by P, N and S_0. In this context, we will often refer to atomic propositions as *state variables*.

Given assumption 2, a set of states R of a labeled state transition graph can be represented by a BDD $R(\bar{v})$. The number of variables in the resulting BDD is equal to $|P|$. For example, the set of initial states S_0 can be represented with a BDD in this way. Similarly, the next state relation N of a labeled state transition graph is represented by a BDD $N(\bar{u}, \bar{v})$ with $2(|P|)$ variables.

5 Finding Reachable States

Many of the ideas used in symbolic model checking can be explained by considering the problem of computing reachable state sets. Let z_0 be a BDD that depends only on variables in $\bar{v}$. We interpret z_0 as representing a set of states, as described in the previous section. We wish to compute a BDD z that represents the states reachable from z_0 via the transitions in the transition relation N. For non-negative integers n, define BDDs z_{n+1} to represent the set of states reachable in $n + 1$ or fewer steps, as follows:

$$z_{n+1} = z_0 \vee \exists \bar{u} \left[z_n \langle \bar{v} \leftarrow \bar{u} \rangle \wedge N(\bar{u}, \bar{v}) \right].$$

To compute z, simply compute z_1, z_2, and so on until you reach a k where $z_k = z_{k+1}$; then $z = z_k$. We call this method of computing z *direct iteration*.

The above computation can be viewed as finding a least fixed point. Let F be a predicate transformer (a function from boolean formulas to boolean formulas) defined by

$$F(z) = z_0 \vee \exists \bar{u} \left[z \langle \bar{v} \leftarrow \bar{u} \rangle \wedge N(\bar{u}, \bar{v}) \right].$$

The set of states reachable from z_0 is then the least fixed point of F. Direct iteration is one method for computing this fixed point. It is easy to modify direct iteration to compute greatest fixed points, as well.

Computing fixed points is a fundamental step used in symbolic model checking, so it is worthwhile to examine its computational complexity. The direct iteration method involves repeatedly computing $F(z_n)$ and checking the equivalence of z_n and z_{n+1} in order to detect if a fixed point has been reached. Since the formulas are represented by BDDs, checking equivalence is either a constant time or linear time operation, depending on the BDD implementation. Most of the computational effort goes into computing $F(z_n)$. The most expensive step of this is computing

$$\exists \bar{u}\,[z_n\langle \bar{v} \leftarrow \bar{u}\rangle \wedge N(\bar{u},\bar{v})].$$

This is an example of computing a *relational product.* Although relational products can computed using the normal BDD algorithms for restriction and boolean connectives, it is much more efficient to use a special purpose algorithm. We assume the variable ordering for the BDDs is of the form $u_1, v_1, \ldots, u_n, v_n$, where the u_k and v_k are the boolean variables in $\bar{u}$ and $\bar{v}$. The algorithm performs the conjunction and the existential quantification over the variables in $\bar{u}$, all in one pass over z_n and $N(\bar{u},\bar{v})$. Thus, the relational product is computed without ever constructing the BDD for

$$z_n\langle \bar{v} \leftarrow \bar{u}\rangle \wedge N(\bar{u},\bar{v}),$$

which may be much larger than the BDD for the relational product itself.

The above discussion of the complexity of finding reachable sets is based on the assumption that a BDD for $N(\bar{u},\bar{v})$ has already been constructed. Section 7 gives more detail about how $N(\bar{u},\bar{v})$ is constructed. In practice, the size of the circuits that can be verified by this method is limited primarily by whether the BDD for $N(\bar{u},\bar{v})$ fits in primary storage. It is possible to represent the transition relation with more than one BDD such that the total number of BDD nodes required is much smaller than if one BDD is used. This is also discussed in Section 7.

5.1 Iterative Squaring

Iterative squaring can be used as another method for computing fixed points that can drastically reduce the number of iterations needed. The direct iteration algorithm computes the least fixed point of F by computing

$$F(\emptyset), F^2(\emptyset), F^3(\emptyset), \ldots, F^n(\emptyset), \ldots$$

until a fixed point is reached (superscript n denotes repeated application).

Iterative squaring depends on noting that the predicate transformer F^2, which is

$$F^2(z) = z_0 \vee \exists \bar{u}\,[z\langle \bar{v} \leftarrow \bar{u}\rangle \wedge (N(\bar{u},\bar{v}) \vee \exists \bar{w}\,[N(\bar{u},\bar{w}) \wedge N(\bar{w},\bar{v})])],$$

is of the same form as F. Therefore, one can compute z by computing the sequence

$$F(\emptyset), F^2(\emptyset), F^4(\emptyset), \ldots, F^{2^n}(\emptyset), \ldots$$

which can converge much more quickly than direct iteration. In effect, iterative squaring first computes the transitive closure of N, which is then used to compute z.

Although iterative squaring can reduce the number of iterations necessary to reach a fixed point exponentially smaller, it can be impractical if the BDDs needed to represent the intermediate computations become too large. Unfortunately, this is appears to be the normal case in practice. In our experience, iterative squaring has been more efficient than direct iteration only on contrived examples.

6 Symbolic Model Checking

Model checking requires determining whether a given CTL formula f is satisfied in the initial states of a labeled state transition graph. In this section, we present a model checking algorithm for CTL that uses BDDs as its internal representation, in order to avoid explicitly enumerating the states of the model. The algorithm is defined by a procedure CHECK which recurses over the structure of the formula.

The procedure CHECK takes the CTL formula to be checked as its one argument. It returns a BDD that depends only on the state variables $\bar{v}$ of the model. The vector $\bar{v}$ has one boolean variable for every atomic proposition in P. The BDD CHECK(f) is true in a given state if and only if the formula f is true in that state. Of course, the output of CHECK depends on the model being checked, so the labeled state transition graph that the formula is checked against is an implicit argument. Recall that given assumptions 1 and 2 a labeled state transition graph is determined by the set of atomic propositions P, the set of initial states S_0, and the transition relation N.

The set P is simply represented by a list of identifiers. The set S_0 is represented by a BDD that depends on the vector of boolean state variables $\bar{v}$; a state is in S_0 if and only if the values of the variables in $\bar{v}$ determined by that state satisfy the BDD. The representation of the transition relation N requires a distinct copy $\bar{u}$ of the state variables. The transition relation is represented by a BDD $N(\bar{v}, \bar{u})$, where $\bar{v}$ is the state before the transition, and $\bar{u}$ is the state after the transition. The state $\bar{u}$ is a successor of $\bar{v}$ whenever the BDD is satisfied.

We define CHECK inductively over the structure of CTL formulas. If f is an atomic proposition p, then CHECK(f) is the BDD that is true if and only if p is true. 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:

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

The definitions of these intermediate procedures are given below. Notice that these intermediate procedures take boolean formulas (represented by BDDs) 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 successor of that state which satisfies f. Thus, we define CHECKEX such that

$$\textsc{Check}\mathrm{EX}(x) \equiv \exists \bar{u}[x\langle \bar{v} \leftarrow \bar{u}\rangle \wedge N(\bar{v}, \bar{u})].$$

Compare the definition of CHECKEX to the relational product in the definition of z_{n+1} in section 5. They are quite similar except that the first case computes the set of states from which a state in x can be reached, while the second computes the states that can be reached from a state in z_n. In other words, CHECKEX performs one step of a backward reachability search instead of a forward reachability search. In spite of this difference, the same basic algorithm described in section 5 for computing relational products can be used here, as well.

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 equation

$$\mathbf{E}[f\ \mathbf{U}\ g] = g \vee (f \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ g]).$$

Using this fixed point characterization, $\textsc{Check}\mathrm{EU}(x, y)$ can be computed by finding the least fixed point z of the equation

$$z = y \vee (x \wedge \textsc{Check}\mathrm{EX}(z)).$$

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 equation

$$\mathbf{EG}f = f \wedge \mathbf{EX}\,\mathbf{EG}f.$$

Thus, CheckEG(x) can be computed by finding the greatest fixed point z of the equation

$$z = x \wedge \text{CheckEX}(z).$$

Again, this fixed point can be computed with either the direct iteration or iterative squaring methods described earlier.

After determining the set S of states that satisfy a formula f, the algorithm checks whether S_0 is a subset of S. If it is, then the model satisfies f.

6.1 Fairness Constraints

Next, we consider the issue of *fairness*. In many cases, we are only interested in correctness along fair computation paths. For example, we may wish to consider only those computations in which some resource that is continuously requested by a process will eventually be granted to the process. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of the logic slightly. A *fairness constraint* can be an arbitrary CTL formula. 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 now restricted to fair paths. In the remainder of this section we describe how to modify the new algorithm to handle fairness constraints. We assume the fairness constraints are given by a set of CTL formulas $C = c_1, \ldots, c_n$.

We define a new procedure CheckFair for checking CTL formulas relative to the fairness constraints in C. 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 **EG**f given fairness constraints C. The formula means that there exists a computation beginning with the current state in which f holds globally (invariantly) and each formula in C holds infinitely often. The set of such states z is the largest set satisfying the following two conditions:

1. All of the states in z satisfy f, and
2. for all $c_k \in C$, for all $s \in z$, there is a path of length one or greater from s to a state satisfying c_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 every formula in C holds infinitely often. This gives us a characterization of CheckFairEG(x) as the greatest fixed point z of the equation

$$z = x \wedge \bigwedge_{k=1}^{n} \text{CheckEX}(\text{CheckEU}(x, z \wedge \text{Check}(c_k))).$$

The above fixed point can be evaluated in the same manner as before. The main difference is that in this case, each time the above expression is evaluated, it causes several CHECKEU calls to be executed, each of which involves computing a fixed point.

The cases of $\mathbf{EX}f$ and $\mathbf{E}[f\ \mathbf{U}\ g]$ under fairness constraints are a bit simpler. Define the set of all states which are on some fair computation as

$$fair = \text{CHECKFAIR}(\mathbf{EG}\,true).$$

Then,

$$\begin{aligned}\text{CHECKFAIREX}(x) &= \text{CHECKEX}(x \wedge fair),\\ \text{CHECKFAIREU}(x,y) &= \text{CHECKEU}(x, y \wedge fair)].\end{aligned}$$

7 Empirical Results

Using BDDs for testing boolean satisfiability is only efficient in a heuristic sense. The problem is, of course, NP-complete in general; the only claim that is made for BDDs is that they perform well for certain useful classes of boolean functions. Likewise, using BDDs for representing state sets in CTL model checking is only of heuristic value, and does not improve the asymptotic complexity of model checking. Therefore, in order to evaluate the method, we need empirical results showing the performance of the method on some problems of practical interest.

We have examined two classes of digital circuits in evaluating the method empirically. The first is a class of simple synchronous pipelines, which include data path as well as control circuitry. The number of states in these systems is far too large to apply traditional model checking techniques, but we have obtained very encouraging results using the BDD method.

The second class of circuits are asynchronous designs with data paths. Convergence of the fixed point expressions in these systems generally requires a much larger number of steps, since a large number of independent asynchronous transitions may be required to complete operations which are synchronized on a single clock transition in a synchronous design. The results on the performance of the BDD method for these circuits are more ambiguous than those for the synchronous pipelines; it is not yet clear to us to what degree the BDD method is applicable to this kind of circuit.

7.1 Synchronous pipelines

The circuits we have used as examples of this category are very simple pipelines that perform three-address logical and arithmetic operations on a register file. The complete state of the register file and pipe registers are modeled. The pipelines have three stages: the operands are read from the register file, then an ALU operation is performed, then the result is written back to the register file. The ALU has a register bypass path, which allows

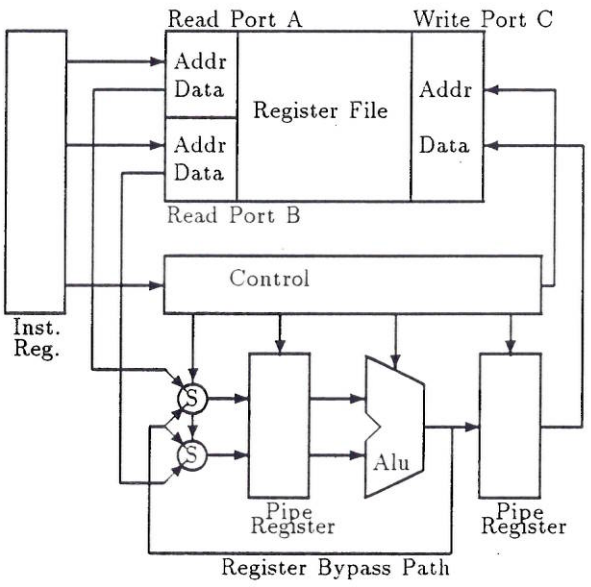

Figure 2: Block diagram of simple pipeline design

the result of an ALU operation to be used immediately as an operand on the next clock cycle, as is typical in RISC instruction pipelines. The inputs to the circuits are an instruction code, containing the register addresses of the source and destination operands, and a STALL signal, which indicates that the instruction stream is stalled. When this occurs, a "no-operation" is propagated through the pipe. A functional block diagram of a typical pipeline is given in figure 2.

Since vectors of boolean values are used to represent binary numbers in these designs, it is useful to introduce some notation for vectors of propositions in logical formulas. First, we define the standard logical and modal operators to operate on vectors of propositions in a component-wise manner. For example,

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \wedge \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix} \equiv \begin{bmatrix} p_1 \wedge q_1 \\ p_2 \wedge q_2 \\ \vdots \\ p_n \wedge q_n \end{bmatrix}$$

and

$$F\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \equiv \begin{bmatrix} Fp_1 \\ Fp_2 \\ \vdots \\ Fp_n \end{bmatrix}$$

In order to deal with the register file, it is also useful to define arrays of propositions (vectors of vectors) and a function $select(v_1, v_2)$ which returns the element of v_1 indexed by the binary number represented by v_2. Note that *select* can be written as a boolean function, in much the same way one might implement a multiplexer using logic gates. It is implemented as a macro in the model checker.

The latency in the example pipelines is three clock cycles. For this reason, the specification of the pipeline cannot be given in a straightforward manner using simply pre- and post- conditions on operations. We can, however, use temporal operators and the above notation to specify the behavior of the pipeline, taking into account the pipe latency. When we specify a register transfer level operation for the pipeline, it is understood that the results of the operation will not affect the register file until three clocks cycles in the future, and the inputs to the operation correspond to the state of the register file two clock cycles in the future. The state of the register file n clock cycles in the future can be expressed using the (vector) modal operator "$\mathbf{X}$", as $\mathbf{X}^n\mathbf{R}$. Thus, taking into account the pipe latency, an RTL specification such as $\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b}$ can be expressed as a temporal formula in the following way:

$$select(\mathbf{X}^3\mathbf{R}, \mathbf{c}) = select(\mathbf{X}^2\mathbf{R}, \mathbf{a}) \oplus select(\mathbf{X}^2\mathbf{R}, \mathbf{b})$$

where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are each bit-fields in the operation code. As similar formulas can be derived for other RTL expressions, we will write RTL syntax in our specifications, with the understanding that it is to be interpreted in the above (temporal) way. Since $\mathbf{X}^n p$ is a path formula and not a state formula, it cannot be evaluated directly by the CTL model checker (which can only evaluate state formulas). We can show, however, that the state of the register file $\mathbf{R}$ two or three clock cycles in the future is uniquely determined by the current state of the system. We can show this by automatically checking the CTL formulas

$$\mathbf{AG}((\mathbf{EX})^2\mathbf{R} \equiv (\mathbf{AX})^2\mathbf{R})$$

and

$$\mathbf{AG}(\mathbf{EX})^3\mathbf{R} \equiv (\mathbf{AX})^3\mathbf{R})$$

Thus, we can substitute the state formula $(\mathbf{EX})^2\mathbf{R}$ for the path formula $\mathbf{X}^2\mathbf{R}$, since the two are equivalent. Likewise, we can substitute $(\mathbf{EX})^3\mathbf{R}$ for $\mathbf{X}^3\mathbf{R}$.

Using the above temporal interpretation for RTL specifications, we write the specification for our simplest pipeline (which has only an exclusive-or instruction) as follows:

$$\mathbf{AG}\neg STALL \Rightarrow (\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b}) \tag{1}$$

and

$$\mathbf{AG}\forall \mathrm{c}'(\mathrm{c} \neq \mathrm{c}' \vee STALL \Rightarrow (\mathbf{R_{c'}} \leftarrow \mathbf{R_{c'}}))$$

The latter formula specifies that non-destination registers do not change, and that if a stall occurs, no registers change.

Table 1 summarizes the results we obtained in verifying a variety of pipelines of this type. We varied the number of bits per register, and the type of operation(s) performed by the ALU, to see how these affected the size of the BDD used to represent the transition relation, the total execution time required to check formula 1, and the total storage used. The most complex pipeline we verified had approximately 5×10^{20} states, which puts it far outside the range of model checkers like the one reported in [5]. It required a BDD with 42,000 nodes to represent the transition relation, and approximately 22 minutes to verify on a Sun 3/60. The most interesting result is that the number of nodes in the transition relation BDD increases only linearly in the number of bits per register. Intuitively, the complexity of the BDD is a function of how much information must be remembered as one passes from one layer of the BDD to the next (i.e., from one variable to the next). In the pipeline examples, the information stored from one bit slice of the data path to the next is simply the state of the control bits plus the value of the ALU carry bit. This amount of information is constant in the number of bits, which explains why the size of the BDD increases linearly in the number of bits.

It is also interesting to note that adding an exclusive-or operation to the addition pipeline roughly doubles the number of nodes in the transition relation characteristic function. This results from the fact the an additional bit has been added to the control information that must be passed down through the data path levels of the BDD, effectively doubling the number of control states. The complexity of control would therefore seem to be a crucial factor in the size of the BDD representation. In addition, if the ALU were able to perform a multiply operation, a barrel shift, or some other complex operation which has more than a constant amount of information passing from one bit position to the next, then the size of the BDD representation would quickly become unmanageable.

7.2 Verifying Asynchronous Circuits

The synchronous pipeline experiments show that it is in fact possible to exploit the regularity of some data paths to construct a compact representation for their state space. Will the same effect be observed for asynchronous

ALU ops	word size	number of registers	BDD size	verification time (secs)
$\oplus$	1 bit	4	2,737	9
$\oplus$	2 bits	4	8,430	46
$\oplus$	3 bits	4	14,123	145
$\oplus$	4 bits	4	19,816	306
$\oplus$	8 bits	4	41,000	1,349
$+$	1 bit	4	2,737	9
$+$	2 bits	4	10,734	45
$+$	3 bits	4	22,276	179
$+$	4 bits	4	33,818	492
$+$	8 bits	4	79,986	3,709
$+, \oplus$	2 bits	4	18,429	188
$+, \oplus$	3 bits	4	36,239	690
$+, \oplus$	4 bits	4	53,924	1,706

Table 1: Performance of BDD model checking algorithm on simple pipelines

circuits? Since the behavior of asynchronous and self-timed circuits is considerably less ordered that that of synchronous circuits, we should expect the complexity of verifying them to be greater. We observed in the previous section that the number of control states had an important impact on the tractability of representing the state space. In self-timed circuits, the number of global control states is generally quite high, due to the loose synchronization between components. On the other hand, a certain kind of regularity can be said to exist in the state space, since many of the possible transitions are mutually commutative—the action of one transition does not affect the enabling conditions of another. The question is, will this effect compensate for the inherently high number of control states in asynchronous circuits? To test this, we applied BDD-based methods to checking hazard freeness in a speed-independent stack element design [15].

We take a different approach to asynchronous circuits (as opposed to synchronous circuits) because of a phenomenon we observed in our experiments that led to a more efficient method. The CTL model checking procedure we used for synchronous circuits begins with a set of states and expands that set by performing a backward breadth-first search though the state graph until a fixed point is reached. This approach provided exactly the result we wanted for specifying the synchronous pipelines, since we wanted to specify the set of states which could reach a state n steps in the future in which a given register bit had a value 1. We discovered, however, that for asynchronous circuits, computing the set of reachable states using forward search progressed much more rapidly than backward search from a set of states representing some failure condition.

Our verification algorithm first computes the set of states H where a hazard can occur, using a method developed by Dill for detecting hazards [12]. The set H is actually represented as an implicit disjunction of BDDs $H_1, \ldots, H_n$. Each H_j represents the set of states where the jth component of the circuit can cause a hazard. Thus, computing H requires no search, and can be done quite quickly. The next step is to compute the set of reachable states, as described in section 5. As this computation is being performed, the algorithm checks that none of the reached states are in H. If no reachable states are in H, the algorithm reports that the circuit is hazard-free.

Another optimization which we have found useful for dealing with asynchronous circuits is to represent the transition relation as a list of characteristic functions, one corresponding to each logic element in the circuit. Since the transitions of each element occur asynchronously, and there is no overall structure or regularity in the organization of the elements, this is a more compact representation than the BDD obtained for the union of all of the separate transition relations.

The performance of the BDD-based verifier on the asynchronous stack is summarized in Table 2. The figure given for the size of the BDD representing the reached state set is the largest for any iteration. This does not in general correspond to the final (and hence largest) set of reached states, since the complexity of the BDD representation is not directly related to the cardinality of the set. The table also gives the number of states reached, and total execution time as a function of the number of data bits. Note that the number of reached states grows by roughly a factor of 10 for each additional data bit, while the number of nodes grow by a factor less than two, and the execution time by a factor between 2 and 3. This is an encouraging result, in that it allows us to check a system with many more states than was previously possibly, and it lends some validity to the conjecture that regularity exists in the state graph of asynchronous circuits. Nonetheless, in this case the BDD algorithm has not given us polynomial complexity in the number of data bits. It has only reduced the base of the exponential. More powerful methods are apparently needed to verify circuits of this type with a large number of data bits.

7.3 Frontier set simplification

In the verification of the stack element, we made use of a technique of Coudert, Berthet, and Madre [11] for simplifying the representation of the *frontier set* (the set of states reached but not yet expanded). This set is the input to the next iteration of the fixed point algorithm, so there is some interest in making its representation as compact as possible. The correctness of the search algorithm is not affected by whether or not the set of reached states passed to the next iteration of the algorithm does or

data bits	approx. gate equivalents	depth of search	BDD size	number of reached states	verification time (secs)
1	30	44	458	272	20
2	50	57	865	1,632	60
3	70	75	1,735	14,696	208
4	90	93	3,101	155,024	726
5	100	111	4,774	$\approx 10^6$	1,878
6	120	129	7,968	$\approx 10^7$	4,588
7	140	147	12,051	$\approx 10^8$	10,416

Table 2: Performance of BDD algorithm for asynchronous stack element

does not contain states which have been previously expanded. When using an explicit representation for the reached state set, it is important not to re-expand any states that have been previously expanded. However, since the complexity of a BDD is not directly related to the number of states it represents, it is often advantageous to re-expand some states if this will reduce the size of the BDD representing the set of states to be expanded. Coudert, Berthet, and Madre describe a method for simplifying the frontier set according to this principle.

We have found that in the case of the stack element, this method reduces the number of nodes in the frontier set BDD by a factor of 2 to 20, depending on the depth of the search. Figure 7.3 shows a graph of the size of the BDD representing the reached states and the simplified frontier set BDD as a function of the number of iterations in the search, for the 6-bit stack element. This graph shows an effect that we have observed a number of times when applying breadth-first search to asynchronous circuits. The graph has a small number of peaks which occur at evenly spaced intervals. We conjecture that these peaks occur at search depths where there is the greatest disparity in the progress of individual elements of the circuits. Completion of a given operation generally occurs after a fixed or nearly fixed number of transitions, even though the order of those transitions is highly arbitrary. These situations correspond to troughs in the graph, where the set of reached states is highly regular, and the set of frontier states is actually very small.

8 Conclusions

As our examples show, the state-explosion problem can sometimes be circumvented by using a symbolic representation for state graphs. When the representation captures the right structural uniformities in the graph, it

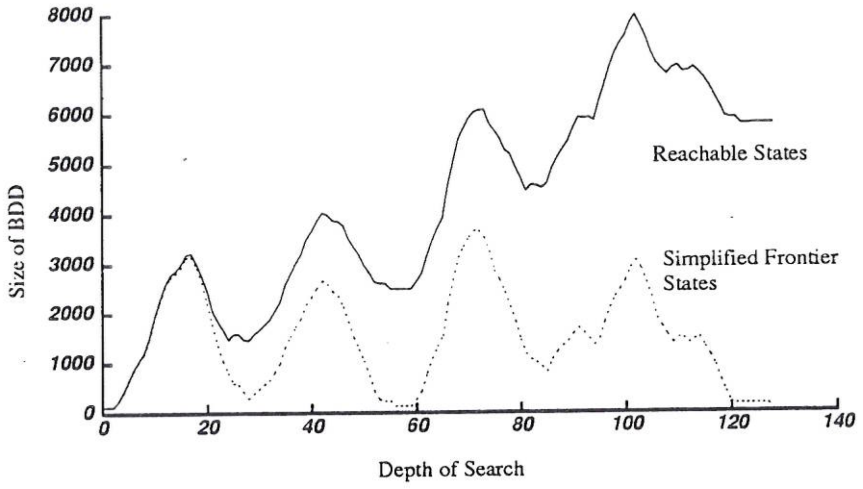

Figure 3: BDD size as a function of search depth for the 6-bit stack.

is much smaller than an explicit table of all of the states. The choice of symbolic representation requires balancing between the expressive power of the representation and the existence of good algorithms for manipulating it.

In our examples, we used binary decision diagrams as the symbolic representation. Results so far indicate that this representation works well much, but not all, of the time. Thus, the method is not necessarily a replacement for brute-force state-enumeration methods, but an alternative that may work efficiently when the brute force methods fail.

Our method is not especially dependent upon the properties of binary decision diagrams. Any representations of boolean functions that supports boolean operations and for which there are good simplification algorithms is a candidate as an internal representation. This is fortunate; because of the importance of boolean functions in CAD for digital systems, a great deal of effort will continue to go into finding better representations and algorithms for manipulating boolean functions. As better representations are developed, they can easily be plugged into our framework to give better verification methods, as well.

Although we have concentrated on temporal-logic model checking, the symbolic state graphs (and specifically binary decision diagrams) can be used in other formalisms for reasoning about sequential and concurrent behavior, such as propositional linear temporal logic and automata on infinite sequences [8].

References

[1] J. Allen and F. T. Leighton, editors. *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference.* MIT Press, 1988.

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

[4] 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.

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

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[7] R. E. Bryant. Verifying a static RAM design by logic simulation. In Allen and Leighton [1], pages 335–349.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and 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.

[9] 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, May 1981. Springer-Verlag.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Sifakis [16].

[12] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In Allen and Leighton [1].

[13] M. Fujita and H. Fujisawa. Specification, verification, and synthesis on control circuits with propositional temporal logic. In J. A. Darringer and F. J. Rammig, editors, *Proceedings of the Ninth International Symposium on Computer Hardware Description Languages and their Applications*, Washington, D.C., June 1989. North-Holland.

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

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

[16] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[17] J. Staunstrup, S. J. Garland, and J. V. Guttag. Localized verification of circuit descriptions. In Sifakis [16].