Automatic verification of sequential circuit designs

By E. M. Clarke[1], J. R. Burch[1], O. Grumberg[2], D. E. Long[1] and K. L. McMillan[1]

[1]*School of Computer Science, Carnegie Mellon, Pittsburgh, Pennsylvania 15213, U.S.A.*

[2]*Computer Science Department, The Technion, Haifa 32000, Israel*

Temporal logic model checking is a method for automatically deciding if a sequential circuit satisfies its specifications. In this approach, the circuit is modelled as a state transition system, and specifications are given by temporal logic formulas. Efficient search algorithms are used to determine if the specifications are satisfied or not. The procedure has been used successfully in the past to find subtle errors in a number of non-trivial circuit designs. Recently, the size of the circuits that can be handled by this technique has increased dramatically. It is now possible to verify transition systems that are many orders of magnitude larger than was previously the case. In this paper, we describe some of the techniques that have made this increase possible. These techniques are based on the use of *binary decision diagrams* to represent transition systems and sets of states.

1. Introduction

Logical errors in sequential circuit designs are an important problem for circuit designers. They can delay getting a new product on the market or cause the failure of some critical device that is already in use. The most widely used method for sequential circuit verification is based on extensive simulation and can easily miss significant errors when the number of possible states of the circuit is very large. Although there has been considerable research on the use of theorem provers, term rewriting systems and proof checkers in hardware verification, these techniques are time consuming and often require a great deal of manual intervention. They have been successfully used for reasoning about data paths, but most logical errors in sequential circuit designs arise because of problems in the control circuitry rather than in the data paths. The research described in this paper is based on an alternative approach called *temporal logic model checking* in which specifications are expressed in a propositional temporal logic and an efficient search procedure is used to determine whether or not the specifications are satisfied. This technique has been used in the past to find subtle errors in a number of non-trivial circuit designs. In this paper we describe some recent extensions of this method that have made it possible to verify much larger circuits than was previously the case.

Temporal logic is a formal system for reasoning about the ordering of events in time without introducing time explicitly. It was originally developed by philosophers for investigating the way that time is used in natural language arguments. Pnueli (1977) was the first to use temporal logic for reasoning about the concurrent

Phil. Trans. R. Soc. Lond. A (1992) 339, 105–120
Printed in Great Britain

programs. Later, Bochmann (1982) and Malachi & Owicki (1981) used this notation for reasoning about circuit designs. However, their correctness proofs were constructed by hand, and they could only handle very small circuits. The introduction of temporal logic model checking algorithms (Clarke & Emerson 1981; Quielle & Sifakis 1981) in the early 1980s allowed this type of reasoning to be automated. *Model checking* is a technique for determining whether a finite state transition system satisfies some formula of temporal logic. In this approach, the transition system represents the circuit to be verified, and the formula represents its specification. Since checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, this technique can be implemented very efficiently. In fact, for computation tree logic (CTL), there is a model-checking algorithm that has complexity linear in both the size of the transition system and its specification (Clarke *et al.* 1986).

Model checking has several important advantages over mechanical theorem provers or proof checkers for sequential circuit verification. The most important is that the procedure is completely automatic. Typically, the user provides a high-level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The procedure is also quite fast, and usually produces an answer in a matter of minutes. Partial specifications can be checked. When a specification is not satisfied, other formulas – not part of the original specification – can be checked to locate the source of the error. Finally, the logic used for specifications can directly express many concurrency properties. Thus, it is unnecessary to introduce special variables to indicate the times at which events occur as is normally the case when some variant of predicate calculus is used as the specification language.

The main disadvantage of the procedure is the *state explosion problem*, which can occur if the device being verified has many components that can make transitions in parallel or if the circuit contains very wide data paths. Early implementations could handle transition systems with only a few thousand states. Therefore, they were only useful for systems with a small number of components. Nevertheless, this was sufficient to find errors in a number of non-trivial, although relatively small, protocols and circuit designs (Browne *et al.* 1986; Dill & Clarke 1986). In the past few years, however, the size of the circuits that can be handled by this technique has increased dramatically. It is now possible to check transition systems that are many orders of magnitude larger in terms of states than the first examples that were tried. In this paper, we describe some of the techniques that have made this increase possible. These techniques use highly efficient data structures to represent the transition systems and sets of states.

In the original implementation of the CTL model-checking algorithm, transition relations were represented explicitly by adjacency lists. The new implementation of the algorithm uses an implicit representation for state transition systems based on *binary decision diagrams* (BDDs) (Bryant 1986). BDDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. The implicit representation used by the new method is quite natural for modelling sequential circuits. Each state is encoded by an assignment of

boolean values to the set of state variables associated with the circuit. The transition relation can, therefore, be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is often succinctly represented by a binary decision diagram. By using this technique, we have been able to handle some examples that would have required the enumeration of 10^{20} states with the original version of the algorithm. A refinement of this technique in which the transition relation is partitioned into several parts has permitted even larger circuits to be checked for correctness. Several other groups (Bryant & Seger 1990; Coudert *et al.* 1990; Touati *et al.* 1990) have obtained comparable results with model-checking algorithms based on BDDs.

Our discussion of these new techniques is organized as follows. The next section explains the model of computation that we use and how we treat both synchronous and asynchronous circuits within this framework. Section 3 contains a brief discussion of BDDs and how circuits are represented using BDDs. The syntax and semantics of CTL are given in §4 along with some typical examples of specifications expressed in this notation. The following section describes the new symbolic model-checking algorithm and demonstrates how the transition relation of a large state transition graph can be partitioned. In §6, we show how model-checking techniques can be used to verify a pipelined ALU and a multiprocessor cache coherency protocol. The paper concludes in §7 with a discussion of some future research directions.

2. Circuits and state transition systems

To reason about the correctness of sequential circuits, it is necessary to have a formal model of circuit behaviour. In this paper, we model circuits as *state transition systems*. We assume that a set V of *boolean state variables* is associated with each circuit. For a synchronous circuit, the set V typically consists of the outputs of all the registers in the circuit. In the case of an asynchronous circuit, there is usually one element of V for each wire. A *state* of the circuit consists of an assignment of truth values to the elements of V. If we identify each state with the set of variables in V that are assigned the value *true*, then the set of all possible circuit states is given by $S = \mathit{Powerset}(V)$. We assume that the set of possible *transitions* between circuit states is given by a relation $R \subseteq S \times S$ and that the circuit has a set of *initial states* $I \subseteq S$ in which it is allowed to start. More formally,

Definition 1. *Let V be a finite set of boolean state variables. A state transition system over V is a triple $M = \langle S, I, R \rangle$, where $S = \mathit{Powerset}(V)$, $I \subseteq S$ is the set of initial states of the system, and $R \subseteq S \times S$ is the transition relation.*

Normally, the transition relation will be given as a boolean formula in terms of the state variables. Two states are related by the transition relation if and only if the truth valuations determined by the states satisfy the boolean formula.

We illustrate how synchronous circuits can be modelled by considering a small example. The circuit in figure 1 is a simple 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 a copy of these variables. We will represent a transition of a circuit by interpreting V to be the present state variables and V' to be next state variables. The transitions of the modulo 8 counter are given by

$$v_0' = \neg v_0, \quad v_1' = v_0 \oplus v_1, \quad v_2' = (v_0 \wedge v_1) \oplus v_2.$$

The above equations can be used to define the relations

$$N_0(V, V') = (v_0' \Leftrightarrow \neg v_0),$$

$$N_1(V, V') = (v_1' \Leftrightarrow v_0 \oplus v_1),$$

$$N_2(V, V') = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2),$$

which describe the constraints that the v_i and v_i' must satisfy in a legal transition. The transition relation is simply the conjunction of these constraints.

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

In the case of a synchronous circuit with n state variables, we let $V = \{v_0, \ldots, v_{n-1}\}$, $V' = \{v_0', \ldots, v_{n-1}'\}$, and assume that for each state variable v_i' there is a function f_i such that

$$v_i' = f_i(V).$$

These equations are used to define the relations

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

As in the case of the modulo 8 counter, the conjunction of these relations forms the transition relation of the circuit.

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

Forming the transition relation for asynchronous circuits is more difficult. For each wire i in the circuit, we derive a transition relation R_i that models how that wire can change. For example, if v_2 is driven by an AND gate with inputs v_0 and v_1, then

$$R_2(V, V') = (v_2' \Leftrightarrow v_0 \wedge v_1).$$

In this paper, we model asynchronous circuits using interleaving semantics. Thus, in our models, at most one wire is allowed to change at a time. If we consider a transition where only wire i may change, then it must satisfy the formula

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

In an asynchronous circuit the wire that changes is determined nondeterministically. Consequently, in this case, the full transition relation is given by the disjunction

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

3. Representing state transition systems symbolically

Circuits with many components may have state transition systems with a very large number of states. An obvious approach for solving the state explosion problem is to develop techniques for succinctly representing large state transition systems. In this section we describe a technique, based on the use of binary decision diagrams, that appears to work quite well in practice.

Figure 1

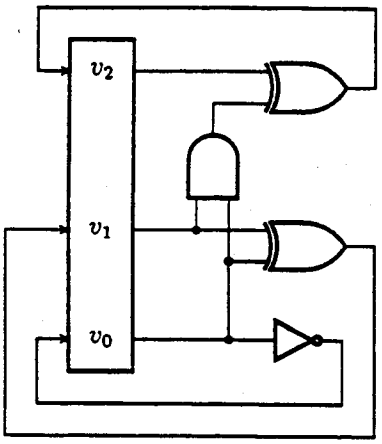

Figure 2

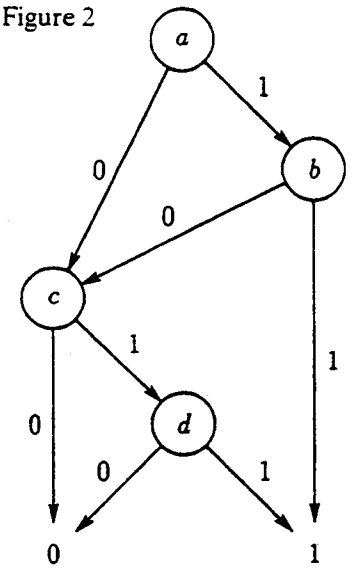

Figure 1. Synchronous modulo 8 counter.
Figure 2. A BDD representing $(a \wedge b) \vee (c \wedge d)$.

(a) *Binary decision diagrams*

Ordered *binary decision diagrams* (BDDs) provide a canonical representation for boolean formulas (Bryant 1986) that is often substantially more compact than conjunctive or disjunctive normal form. Since formulas expressed in this manner can also be manipulated very efficiently in programs, BDDs have become widely used in 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 2. 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 $\{a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1\}$ leads to a leaf node labelled 1, hence the formula is true for this assignment.

Bryant showed that if the variable ordering is fixed, there is a unique BDD for every boolean formula. The size of the BDD depends critically on the variable ordering. Bryant also devised efficient algorithms for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for formulas f and g. The only additional operations that we require for the algorithms that follow are quantification over boolean variables and substitution of variables. It is straightforward to give an algorithm that finds 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 $f|_{v=0} \vee f|_{v=1}$. Substitution of a variable w for a variable v in a formula f can be accomplished using quantification:

$$\exists v[(v \Leftrightarrow w) \wedge f].$$

BDDs can also be viewed as a form of deterministic finite automata (Kimura & Clarke 1990). An n-argument boolean function can be identified with the set of

strings in $\{0,1\}^n$ that represent valuations where the function is true. 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, conjunction corresponds to language intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages. Based on this approach, it is relatively easy to develop a parallel algorithm for constructing BDDs. Kimura & Clarke (1990) give such an algorithm and describe its performance on a 16 processor Encore Multimax. The execution statistics that have been obtained for a number of examples show that this algorithm achieves a high degree of parallelism. In fact, on many examples it exhibits essentially linear speedup as the number of processors is increased.

(b) *Describing transition systems with binary decision diagrams*

In principle, it is easy to represent a set of system states by a BDD. Consider a circuit in which V is the set of state variables. A state is determined by an assignment of either 0 or 1 to each variable in V. Given such a truth valuation, it is possible to write a boolean expression that is true for exactly that valuation. For example, given $V = \{v_0, v_1, v_2\}$ and the valuation $\{v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\}$, we obtain the boolean formula $v_0 \wedge v_1 \wedge \neg v_2$. This formula can, of course, be represented using a BDD. In general, a boolean formula may be true for many different truth 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 tend to be much more concise than traditional techniques for representing sets of states. In the remainder of the paper, we will denote sets of states with S. We denote the BDD for the set S by $S(V)$, where V is the set of variables that the BDD depends on.

In addition to representing sets of system states, we must be able to represent the transition relation. To do this, we extend the idea used above. A valuation for the variables in V and V' can be viewed as designating a pair of states that determine a transition. We can represent sets of such valuations using BDDs as above. For the circuit example given in §2, the transition relation is represented by the BDD for the formula $N(V, V')$.

4. The logic CTL

To specify properties of state transition systems, we need a logic that can describe the relative ordering of events in time. An obvious way of developing such a logic is to introduce special variables into a predicate logic that represent the times at which events occur. Decision procedures for even the simplest fragments of such logics have high complexity, however. Hence, it is difficult to develop fully automatic verification tools if this approach is used. Instead, we use a propositional, temporal logic called *computation tree logic* (CTL) (Clarke *et al.* 1986) to specify properties of state transition systems. Special operators are included in CTL for reasoning about the ordering of events in time without introducing time explicitly. We begin this section with an informal description of CTL and some examples of how properties that arise in sequential circuit verification can be expressed in this logic.

Suppose that we wish to reason about a state transition system M. If some state of M is selected as the *initial* state, then M can conceptually be unwound into an

infinite tree with that state as its root (see figure 3). Since paths in the tree represent possible behaviours or computations of the transition system, we will refer to the infinite tree obtained in this manner as the *computation tree* of M. CTL contains special operators for describing computation trees. Formulas in CTL are built from atomic propositions (one for each state variable in the circuit), boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\Leftrightarrow$ and $\oplus$), and *temporal operators*. Each operator consists of two parts: a path quantifier (**A** or **E**) and a temporal modality (**F**, **G**, **X**, or **U**). The quantifier indicates whether the operator denotes a property that should be true of all paths from a given state or whether the property need only hold on some path. The modalities describe the ordering of events in time along a computation path and have the following intuitive meanings:

1. $\mathbf{F}\varphi$ ('φ holds sometime in the future') is true of a path if there exists a state on the path for which the formula φ is true.
2. $\mathbf{G}\varphi$ ('φ holds globally') means that φ is true at every state on the path.
3. $\mathbf{X}\varphi$ ('φ is holds next time') means that φ is true at the second state on the path, i.e. the state immediately following the present state.
4. $\varphi\mathbf{U}\psi$ ('φ holds until ψ holds') means that there exists some state on the path such that ψ is true at that state, and that for all preceding states, φ is true.

Each formula of the logic is either true or false in a given state. An atomic proposition is true in a state if the state variable corresponding to the proposition is true in the state. The truth of a formula built from boolean connectives depends on the truth of its subformulas in the usual way. A formula whose top level operator is a temporal operator with a universal (existential) path quantifier is true whenever all paths (some path) starting at the state have the property required by the operator's modality. A formula is true of a circuit if it is true for all the initial states of the circuit. The following examples illustrate the expressive power of the logic:

1. $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: it is always the case that if the signal *req* is high, then eventually *ack* will also be high.
2. $\mathbf{AG}(pending \rightarrow \mathbf{EX}\ completed)$: it is always the case that if *pending* is high, then it is possible for *completed* to be high in the next state.
3. $\mathbf{EF}(started \wedge \neg ready)$: it is possible to get to a state where *started* holds but *ready* does not hold.
4. $\mathbf{AG}\,\mathbf{AF}\ enabled$: *enabled* holds infinitely often on every computation path.
5. $\mathbf{AG}\,\mathbf{EF}\ restart$: from any state, it is possible to get to the *restart* state.
6. $\mathbf{AG}(send \rightarrow \mathbf{A}(send\mathbf{U}recv))$: it is always the case that if *send* occurs, then eventually *recv* is true, and until that time, *send* must remain true.

We conclude this section with a more precise description of CTL to provide a rigorous basis for explaining the decision procedure in §5. Let $M = \langle S, I, R\rangle$ be a state transition system. A *path* in M is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for every $i \in \mathbb{N}$, $R(s_i, s_{i+1})$. We write $M, s \models \varphi$ to indicate that the formula φ is true in state s of M. This relation is defined inductively as follows:

1. If φ is the atomic proposition corresponding to the state variable v, then $s \models \varphi$ if and only if $v \in s$.
2. $s \models \neg\varphi$ if and only if it is not the case that $s \models \varphi$. $s \models \varphi \wedge \psi$ if and only if $s \models \varphi$ and $s \models \psi$. The other propositional connectives are handled similarly.
3. $s \models \mathbf{EX}\varphi$ if and only if there exists a path $\pi = s_0 s_1 s_2 \ldots$ starting at $s = s_0$, such that $s_1 \models \varphi$.
4. $s \models \mathbf{EG}\varphi$ if and only if there exists a path π starting at s such that for every state s' on π, $s' \models \varphi$.

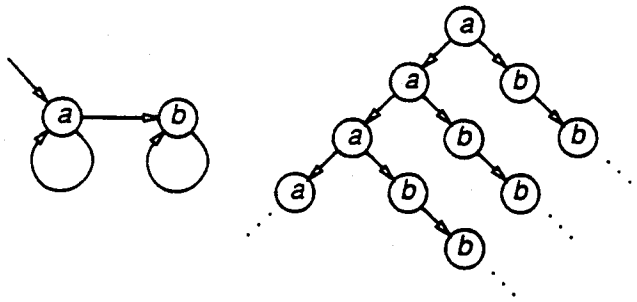

Figure 3. State transition graph and corresponding computation tree.

5. $s \models \mathbf{E}(\varphi \mathbf{U} \psi)$ if and only if there exists a path $\pi = s_0 s_1 s_2 \ldots$ starting at $s = s_0$ and some $i \geqslant 0$ such that $s_i \models \psi$ and for all $j < i, s_j \models \varphi$.

The semantics of the temporal operators with universal path quantifiers can be defined in terms of those given above. For example, $\mathbf{AG}\varphi$ is equivalent to $\neg \mathbf{E}(\mathit{true}\mathbf{U} \neg\varphi)$ and $\mathbf{A}(\varphi \mathbf{U} \psi)$ is equivalent to $\neg\mathbf{E}(\neg\psi \mathbf{U}(\neg\psi \wedge \neg\varphi)) \wedge \neg \mathbf{EG} \neg\psi$. Finally, we will write $M \models \varphi$ to indicate that every initial state of M satisfies the formula φ.

5. Symbolic model checking

Temporal logic model checking algorithms (Clarke & Emerson 1981: Clarke *et al.* 1986: Lichtenstein & Pnueli 1985) may be used to determine automatically whether a state transition system satisfies a temporal logic formula. The algorithm described below is an efficient model checking algorithm for CTL that makes use of the symbolic representation of circuits and sets of states described previously. Given a formula φ, the algorithm determines the set of states (represented as a BDD) where each subformula of φ (including φ itself) is true. It does this in a bottom up fashion starting from the atomic propositions in the formula.

1. For an atomic proposition corresponding to the state variable v, the set of states where the proposition is true is described by the boolean formula v.

2. For a conjunction of formulas $\varphi \wedge \psi$, the set of states where the conjunction is true is the intersection of the set of states where φ is true and the set of states where ψ is true. Given BDDs $S_\varphi(V)$ and $S_\psi(V)$ corresponding to the latter, the BDD corresponding to the conjunction is simply $S_\varphi(V) \wedge S_\psi(V)$. Other boolean operations on formulas are handled similarly.

3. For the formula $\psi = \mathbf{EX}\varphi$, we first find the BDD $S_\varphi(V)$ representing the set of states where φ is true. The states where $\mathbf{EX}\varphi$ is true are those that have a successor for which φ is true. If $N(V, V')$ is the formula for the transition relation, then we can write a boolean formula for $\mathbf{EX}\varphi$ by quantification over the state variables in V:

$$S_\psi(V) = \exists_{v \in V'} [S_\varphi(V') \wedge N(V, V')].$$

Note that this results in n nested quantifications if there are n variables in V. Forming the conjunction of $S_\varphi(V')$ with $N(V, V')$, gives a BDD that represents those pairs of states s and t for which there is a transition from s to t and φ is true in t. Quantifying out the variables in V' gives a BDD that represents those states s which have *some* successor satisfying φ.

4. For a formula such as $\mathbf{EF}\varphi$, we use a fixed point characterization of the

temporal operator to compute a BDD representing the set of states for which $\mathbf{EF}\varphi$ is true. $\mathbf{EF}\varphi$ is the least fixed point (under the inclusion ordering) of

$$\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\mathbf{EF}\varphi.$$

Given such a fixed point characterization, we can find the appropriate solution by simply iterating. We begin with the set of states where φ is true represented as the BDD $S_\varphi(V)$. We then perform the **EX** operation as above on this set of states and union the result with $S_\varphi(V)$. This gives a new BDD $S(V)$ describing those states which either satisfy φ or which can reach a state satisfying φ in one step. We then repeat the process; since the set of possible states is finite, we will eventually reach the desired fixed point. Also note that detecting this condition is very efficient because BDDs provide a canonical form for representing sets of states. Formulas with other temporal operators such as **EG** can be handled in a similar manner.

5. The formulas with universal path quantifiers can be rewritten in terms of formulas with existential path quantifiers and boolean operations.

The algorithm can be extended to handle *fairness constraints* as described by Clarke *et al.* (1986). In addition, the model-checking algorithm will give a counterexample trace (if this is possible) when it finds that a formula is false. This feature is particularly useful in debugging complex circuits.

Several parts of the algorithm involve performing computations of the following form:

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

This expression is called a *relational product*, and it forms the basis for many verification techniques that use symbolic methods. Thus, it is crucial to be able to perform this step efficiently. For example, a special algorithm is typically used which performs this operation without building the BDD for $S(V') \wedge N(V, V')$, which would often be impractically large. Unfortunately, the BDD $N(V, V')$ itself is often very big, and being forced to construct this BDD has been the major stumbling block in trying to verify complex circuits. In the remainder of this section, we describe techniques for overcoming this problem by using more than one BDD to represent N.

(*a*) *Asynchronous circuits*

Since the transition relation for an asynchronous circuit is a disjunction of relations, the relational product computed is of the form

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

This relational product can be computed without ever constructing the BDD for the full transition relation by rewriting the formula as follows.

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

Thus we are able to reduce the problem of computing the full relational product one of computing a series of relational products involving relatively small BDDs. This technique was used to verify the design of an asynchronous stack in (Burch *et al.* 1990*a*). Larger asynchronous circuits can be verified by the same method.

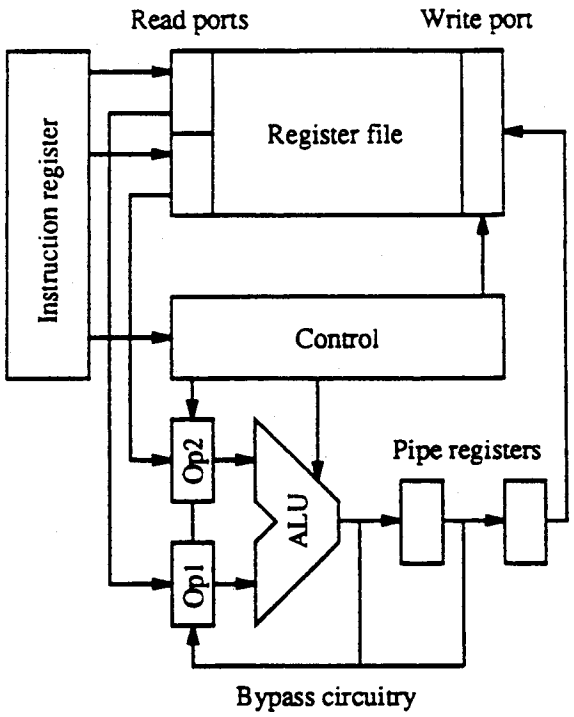

Figure 4. Pipeline circuit block diagram.

(*b*) *Synchronous circuits*

Since the transition relation for a synchronous circuit is a conjunction of relations, the relational product computed has the form

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

The technique used for asynchronous circuits cannot be applied here because existential quantification does not distribute over a conjunction. However, a different technique can be used in this case to construct the BDD for the full transition relation. We use the modulo 8 counter described earlier to illustrate this technique. Since the counter has three state variables, the relational product is given by

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

Since conjunction is associative, and the ordering of existential quantification does not matter, we can rewrite this as

$$\exists v_2' \exists v_1' \exists v_0' [((S(V') \wedge N_0(V, V')) \wedge N_1(V, V')) \wedge N_2(V, V')]. \tag{2}$$

Next, we make use of the fact that subformulas can be moved out of the scope of an existential quantifier if they do not depend on any of the variables being quantified. Since $N_2(V, V')$ does not depend on v_0' or v_1', we obtain

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

Since $N_1(V, V')$ does not depend on v_0', we can apply this transformation one more time by writing

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

We can now compute the relational product in equation (1) by starting with $S(V')$ and at each step conjoining 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.

The ordering chosen for the conjuncts in equation (2) can have a major impact on the performance of the algorithm. We wish to order the $N_i(V, V')$ so that the variables in V' can be quantified out as quickly as possible, and 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 number of new state variables v_i' in the intermediate BDDs is independent of the order of the $N_i(V, V')$. However, the number of old state variables v_i depends on the order, and is minimized by the order given in equation (2).

6. Examples

In this section, we discuss two examples that illustrate the application of symbolic model checking. The first example is a simple data pipeline of the type commonly used in RISC processors, and the second is a distributed cache consistency protocol.

(a) *Data pipeline*

The data pipeline circuit performs three-address arithmetic and logical operations on operands stored in a register file. 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.

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 which 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. 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.

A more detailed CTL specification for the pipeline is given elsewhere (Burch *et al.* 1990*a*); we provide a brief summary here. In particular, the pipeline considered in this section performs only addition operations. The main correctness condition states that when an instruction is issued, then $s+2$ cycles later, the destination register contains the appropriate value. The value that the destination register should contain depends on the results from instructions which are still executing since results from these instructions may be bypassed to the ALU operand registers. We can take this into account by noting that these results will all be reflected in the register file $s+1$ cycles after the instruction is issued. Thus, after $s+2$ cycles, the value in the destination register should be the sum of those values that are in the source registers

$s+1$ cycles after the instruction is issued. Let $reg_{j,i}$ denote the state variable for bit i of register j. Since the pipeline is deterministic and every state has a successor, we can express the value of $reg_{j,i}$ k cycles in the future by the CTL formula

$$\overbrace{\mathbf{AX}\,\mathbf{AX}\cdots\mathbf{AX}}^{k} reg_{j,i}.$$

(We abbreviate this as $\mathbf{AX}^k reg_{j,i}$.) When there are two registers, bit i of the destination register $s+2$ cycles in the future is given by

$$(\neg destaddr \wedge \mathbf{AX}^{s+2} reg_{0,i}) \vee (destaddr \wedge \mathbf{AX}^{s-2} reg_{1,i}).$$

We abbreviate this formula as $dest_i$. Similar formulas are used to specify the source operand values. These formulas are used to write formulas sum_i that correspond to the individual bits in the sum of the source operands. Then the fact that either the pipeline is stalled or bit i of the destination register gets bit i of the sum of the source operands is given by $\mathbf{AG}(\neg stall \rightarrow (sum_i \Leftrightarrow dest_i))$.

From the diagram of the circuit, we see that it decomposes naturally into pieces. We used this decomposition with the techniques described in §5 to reduce the size of the BDDs constructed during the verification. 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 the individual pipe registers, without increasing the number of BDD nodes required; we did this to decrease overhead. The final partitioned transition relation consisted of the following pieces: (1) control logic; (2) pipe registers; (3) the first ALU operand register; (4) the second ALU operand register; and (5) one piece for each general register. This ordering was also used in computing the relational product as described in §5*b*.

We performed the tests described above by using a CTL model checker written mostly in LISP. The actual BDD manipulation routines are written in C and are based on a package by Brace *et al.* (1990). The model checker was run on a Sun 4. We experimented with several different pipeline configurations. The largest of these had eight registers, each 32 bits wide, and two pipe registers, giving a total of 406 state variables and more than 10^{120} reachable states. The verification took 4 h and 20 min of CPU time. For all of the configurations that we considered, the verification time grew polynomially in the *number of pipeline components*. Moreover, when the relational product was computed with the ordering described above, the sizes of the intermediate results increased monotonically during each step. In other words, partitioning the transition relation did not result in having to manipulate larger BDDs than would have been necessary with a monolithic 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 problems that can be handled.

(*b*) *Distributed cache protocol*

The symbolic model-checking technique has been applied to the verification of the cache consistency protocol of the Encore Gigamax multiprocessor (McMillan & Schwalbe 1991). The Gigamax is a distributed, shared memory multiprocessor, in which the processors are grouped into clusters. Each cluster has a local bus, and uses bus snooping (Archibald & Baer 1986) to maintain consistency within the cluster. In addition, each cluster has an interface called a UIC, which links the cluster into a network. The UIC keeps the caches in the cluster consistent with the rest of the

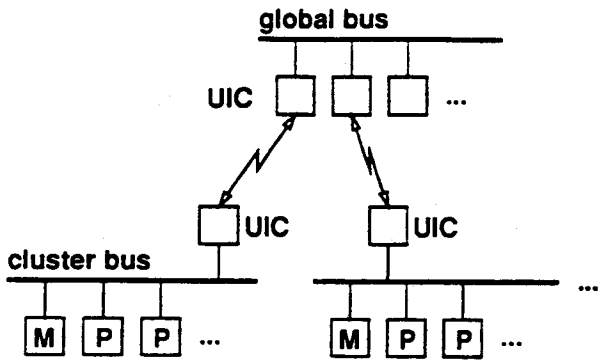

Figure 5. Gigamax memory architecture.

network by acting as both a bus snooper and a bus master on behalf of the remote clusters. The UIC has a table which keeps track of the remote status of all addresses from the local main memory. This allows the UIC to intervene in bus transactions which affect remotely owned addresses, and to send appropriate invalidation or call back requests to the network. The network is organized into a hierarchy, as depicted in figure 5. The global bus, at the top of the hierarchy, has UICs connected to each cluster. Each UIC on the global bus records the state of all cache lines which are held in the cluster to which the UIC is connected. This information makes directory pointers in main memory unnecessary.

An abstract, architectural level model of the Gigamax was constructed – essentially a system of communicating finite state machines. The model was checked for the following properties (here, p_n denotes the nth cache in the system):

1. Single-line data consistency:

$$consistent(p_n, p_m) \equiv ((p_n \cdot valid \wedge p_m \cdot valid) \Rightarrow p_n \cdot data = p_m \cdot data)$$

$$\forall p_n, p_m : \mathbf{AG}\,\mathbf{AF}(consistent(p_n, p_m) \vee \exists p_l : p_l \cdot write).$$

2. Absence of deadlock:

$$\forall p_n : ((\mathbf{AG}\,\mathbf{EF}\, p_n \cdot shared) \wedge (\mathbf{AG}\,\mathbf{EF}\, p_n \cdot owned)).$$

3. Correctness of diagnostics:

$$\forall p_n : \mathbf{AG} \neg p_n \cdot error.$$

The first of these states states that if no cache is written, then all pairs of caches eventually become consistent. The second states that it is always possible for a cache to enter the shared (readable) state, and it is always possible for a cache to enter the owned (writable) state. The third states that the diagnostic system never reports an error (under normal operation).

The symbolic model checker SMV performed an exhaustive search of the model's state space without explicitly constructing the global state graph. This was important, since the number of states in the model was as high as 10^{15}, depending on the number of clusters, and the number of processors in each cluster. We performed the verification on a Sun 3/60. For a model with two clusters, and six processors per cluster, the number of states was 2.0×10^{15} and the execution time was $8\frac{1}{2}$ min. The number of BDD nodes representing the largest approximation to the reached state set was 3293, and the number of nodes representing the transition relation was 37 556. The transition relation was partitioned disjunctively, as described in §5, with one disjunct for each of the three busses. The execution times grew cubically with the number of caches per cluster.

Checking the three specifications exposed a number of subtle errors in the design that were not found in simulation. These errors were usually caused by events, e.g. cache misses and message arrivals, occurring out of the normal sequence anticipated by the designers. Since they typically resulted from a highly improbable confluence of events, the likelihood of finding them by random simulation methods was low. For example, one deadlocked state uncovered by the model checker required a minimum of 13 steps to reach. As the design evolved to correct these errors, the model was easily adapted, and quickly provided an analysis of any new errors introduced by design changes.

7. Conclusion

The techniques described in this paper have already been used to find non-trivial errors in circuit designs. We are currently investigating the correctness of other hardware systems with realistic complexity like the cache coherency protocol used in the IEEE Futurebus standard. We believe that our current model checking algorithm works sufficiently well in practice to be of use in industry and have begun collaboration with Intel Israel to develop a version of the program that will be suitable for circuit designers. Since model checking avoids the construction of complicated proofs and provides a counterexample trace when some specification is not satisfied, we believe that circuit designers will find this technique relatively easy to learn and use. We plan to adapt the CTL verifier for use with VHDL and Verilog, since it now appears that these hardware description languages will become widely used in industry. We hope to have a prototype verification system for one of these languages running by the middle of 1992.

Nevertheless, we believe that there are a number of ways our CTL model checker can be improved to make it easier to use by engineers. Some of these improvements involve relatively straightforward extensions of current system. For example, an obvious problem with the current system is how to make the specification language more expressive and easier to use. Some type of *timing diagram* notation may be more natural for engineers than CTL. It may be possible either to translate timing diagrams systematically into temporal logic formulas or to check them directly by using an algorithm similar to the one used by the model checker. A similar problem arises in finding a good way to display the counterexamples that are generated when a formula is not true. This feature is invaluable for actually finding the source of a subtle error in a circuit design. However, our current system just prints out a path in the state transition graph that shows how the error occurs. It is easy to imagine more perspicuous ways of displaying this information.

Other extensions require more theoretical research. An obvious direction for research is to develop even more concise techniques for representing boolean functions. Our verification method is not especially dependent upon the properties of binary decision diagrams. In fact, any representation of boolean functions that supports boolean operations and for which there are good simplification algorithms is a candidate for such a representation. As better representations are developed, they can easily be incorporated into our model checking algorithms. Moreover, it should be possible to adapt the methods that we use for representing transition relations and solving fixed point equations to other formalisms for reasoning about finite state concurrent systems like linear temporal logic, automata on infinite sequences, and various bisimulation relations from process algebra (Burch *et al.* 1990*b*).

Ultimately, we expect to obtain more benefit from the use of abstraction and compositional reasoning techniques. Much additional research is needed on both of these topics. It may be possible to reason about certain types of infinite state systems by using abstraction techniques. We also plan to investigate the use of symmetry in reasoning about circuits. Most large circuits are highly symmetric. This symmetry should be reflected in the state transition graph of the circuit. We believe that it may be possible to exploit this observation to avoid searching the entire state space of the circuit. Another important problem concerns the use of induction with model checking. Many circuit designs are *parametrized*. For instance, a hardware stack may be parametrized by its length and the size of elements it contains. Model checking can be used to show the correctness (or incorrectness) of specific instances of such a circuit, but some type of inductive argument appears necessary to establish the correctness of the general design. There has already been some research on this problem (Browne *et al.* 1989; Kurshan & McMillan 1989), but more work is needed.

We are currently attempting to extend our methodology to real-time concurrent systems. Such systems are particularly difficult to verify because their correctness depends on the actual times at which events occur. We are developing an automatic verifier for real-time systems. The verifier uses a discrete time model that represents the passage of time with clock ticks. The formal relationship between this model of time and a continuous time model is being investigated by Burch (1991). We believe we can show that this model is a conservative approximation of a more realistic continuous time model. The verifier will also use algorithms based on binary decision diagrams. We expect this will help avoid the state explosion problem and allow the verification of larger systems than would be the case with explicit state enumeration algorithms.

Finally, we believe that it is important to investigate how model checking techniques can be combined with theorem proving. We suspect that ultimately both model checkers and theorem provers will be needed to establish the correctness of complex circuits. Theorem provers seem necessary for reasoning about those parts of a complex microprocessor like the floating point arithmetic unit that require relatively deep mathematical knowledge. On the other hand, it seems unlikely that existing theorem proving systems will surpass model checking techniques for reasoning about complex hardware controllers in the near future. The problem is how to combine the two very different styles of reasoning into a single framework so that a user can smoothly integrate the results obtained by each.

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; the National Science Foundation under Contract no. CCR-9005992; and the U.S.–Israeli Binational Science Foundation.

References

Archibald, J. & Baer, J. L. 1986 Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Computer Syst.* **4**, 273–298.

Bochmann, G. V. 1982 Hardware specification with temporal logic: an example. *IEEE Trans. Computers* **C-31**(3), 223–231.

Brace, K. S., Rudell, R. L. & Bryant, R. E. 1990 Efficient implementation of a BDD package. In DAC90 (1990).

Browne, M. C., Clarke, E. M., Dill, D. L. & Mishra, B. 1986 Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Computers* **C-35**, 1035–1044.

Browne, M. C., Clarke, E. M. & Grumberg, O. 1989 Reasoning about networks with many identical finite state processes. *Inform. Computat.* **81**(1).

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

Bryant, R. E. & Seger, C.-J. 1990 Formal verification of digital circuits using symbolic ternary system models. In Kurshan & Clarke (1990).

Burch, J. R. 1991 Automatic symbolic verification of real-time concurrent systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA 15213. (In preparation.)

Burch, J. R., Clarke, E. M., McMillan, K. L. & Dill, D. L. 1990*a* Sequential circuit verification using symbolic model checking. In DAC90 (1990).

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

Clarke, E. M. & Emerson, E. A. 1981 Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of programs: workshop* (ed. D. Kozen), vol. 131 of *Lecture Notes in Computer Science*. Yorktown Heights, New York: Springer-Verlag.

Clarke, E. M., Emerson, E. A. & Sistla, A. P. 1986 Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.* **8**, 244–263.

Coudert, O., Madre, J. C. & Berthet, C. 1990 Verifying temporal properties of sequential machines without building their state diagrams. In Kurshan & Clarke (1990).

DAC90 1990 *27th ACM/IEEE Design Automation Conference*.

Dill, D. L. & Clarke, E. M. 1986 Automatic verification of asynchronous circuits using temporal logic. *IEE Proc.* E **133**(5).

Kimura, S. & Clarke, E. M. 1990 A parallel algorithm for constructing binary decision diagrams. In *Proceedings: IEEE International Conference on Computer Design*.

Kurshan, R. & Clarke, E. M. (eds) 1990 *Workshop on Computer-Aided Verification*. New Brunswick, New Jersey: Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). Technical Report 90-31.

Kurshan, R. P. & McMillan, K. L. 1989 A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press.

Lichtenstein, O. & Pnueli, A. 1985 Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles on Programming Languages*.

Malachi, Y. & Owicki, S. S. 1981 Temporal specifications of self-timed systems. In *VLSI systems and computations* (ed. H. T. Kung, B. Sproull & G. Steele), pp. 203–212.

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

Pnueli, A. 1977 The temporal semantics of concurrent programs. In *18th Symposium on Foundations of Computer Science*.

Quielle, J. P. & Sifakis, J. 1981 Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*.

Touati, H. J., Savoj, H., Lin, B., Brayton, R. K. & Sangiovanni-Vincentelli, A. 1990 Implicit state enumeration of finite state machines using BDD's. In *IEEE International Conference on Computer-Aided Design*.

Discussion

P. Thompson (*Inmos Ltd, Bristol, U.K.*). Is the partitioning of the transition relation automated or performed by hand?

E. M. Clarke. At present, the partitioning is done by hand. In the examples that we have considered, finding a good partition has been fairly simple. We believe that this procedure is potentially automatable. However, we do not have a general algorithm for this problem at the present time.