Representing Circuits More Efficiently in Symbolic Model Checking

J. R. Burch E. M. Clarke D. E. Long
School of Computer Science
Carnegie Mellon University

Abstract

We significantly reduce the complexity of BDD-based symbolic verification by using *partitioned transition relations* to represent state transition graphs. On an example pipeline circuit, this technique reduced the verification time by an order of magnitude and the storage requirements for the transition relation by two orders of magnitude. We were also able to handle example pipelines with over 10^{120} reachable states.

1 Introduction

Although methods for verifying sequential circuits by searching their state transition graphs have been investigated for many years, it is only recently that such methods have begun to seem practical. Before, the largest circuits that could be verified had about 10^6 states. Now it is easy to check circuits that have many orders of magnitude more states [5, 6, 7, 9]. The reason for the dramatic increase is the use of special data structures such as binary decision diagrams (BDDs) [4] for encoding the state transition graphs of such systems.

In this paper, we show how to process state transition graphs more efficiently than in our previous work [6, 7]. Our new approach involves using multiple BDDs which are implicitly conjuncted to represent the graphs. We call this representation a *partitioned transition relation*. The BDDs that make up the partitioned transition relation are derived in a natural way from the structure of the circuit being verified. We illustrate the power of the technique by verifying a pipeline circuit [6] against a specification given in the temporal logic CTL [8]. For a pipeline with 4 registers, each 32 bits wide, the partitioned transition relation required less than 2,500 BDD nodes, while our previous approach of using a single BDD required nearly 340,000 nodes, a savings of nearly a factor of 140. On a Sun 4, the verification time improved from approximately 14,000 seconds (projected) to 995 seconds, a factor of about 14. We were also able to handle example pipelines with over 10^{120} reachable states.

There are several other methods that use BDDs in the verification of sequential circuits. Bryant and Seger [5] use a symbolic switch-level simulator to check pre- and post-conditions specified in a restricted form of temporal logic. The logic allows boolean conjunction and the next time modality (X). Bose and Fisher [2] show how to verify pipeline circuits with respect to a simpler abstract model by means of a representation function. Coudert, Berthet, and Madre describe a system for showing equivalence between deterministic finite automata [9]. Their system performs a symbolic breadth-first search of the state space reachable by of the product of the two automata. More recently [10], they have extended their method to handle properties expressed in the temporal logic CTL [6, 8]. None of these methods, except for the recent work by Coudert *et al.*, can be used to verify liveness properties. More importantly, none of these methods can easily handle nondeterministic systems. With transition relations, it is very natural to model examples like the cache coherency protocol for the Encore Gigamax, which McMillan has recently investigated [12]. A major feature of the Gigamax architecture is an asynchronous, and hence nondeterministic, interconnection network. The use of abstraction to hide certain details of the cache replacement policy also gives rise to nondeterminism in this example.

Our paper is organized as follows. Section 2 reviews symbolic model checking. Section 3 describes how transition relations can be partitioned according to the structure of circuits. We use these partitioned transition relations in section 4 to give improved verification algorithms. Section 5 analyzes the performance of our new techniques on the pipeline circuit mentioned above. The paper concludes in section 6 with a discussion of our results and those of other researchers.

2 Symbolic model checking

In this section, we give an overview of how we represent circuits and sets of circuit states using BDDs and how we specify and verify properties of circuits. These techniques have been described in greater detail in earlier papers [6, 7].

Given a circuit, let V be its set of boolean state variables.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, and in part by the National Science Foundation under contract numbers CCR-8722633 and MIP-8858807.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-395-7/91/0006/0403 $1.50

We identify a boolean formula over V with the set of valuations which make the formula true. A valuation of the variables corresponds in a natural way to a state of the circuit; hence the formula may be thought of as representing a set of circuit states. The BDD for the formula is in practice a concise representation for this set of states. In the remainder of the paper, we will denote sets of states with S's. We denote the BDD representing the set S by $S(V)$, where V is the set of variables that the BDD depends on. In addition to representing sets of states of a circuit, we must be able to represent which transitions the circuit can make. To do this, we use a second set of variables V'. A valuation for the variables in V and V' can be viewed as designating a pair of states in the circuit, and we can represent sets of pairs using BDDs as above. We will refer to sets of pairs of states as transition relations. If N is a transition relation, then we write $N(V,V')$ to denote the BDD that represents it.

We use a propositional temporal logic called CTL [8] to specify properties of circuits. The formulas in CTL are built from atomic propositions (one for each state variable of the circuit), boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\Leftrightarrow$ and $\oplus$), and temporal operators which are used to specify sequential behaviors. Some of the temporal operators are $\forall\mathbf{X}\,\varphi$, which means that φ is true in all immediate successor states, $\forall\mathbf{G}\,\varphi$, which means that φ is true for all reachable states, and $\forall(\varphi\,\mathbf{U}\,\psi)$, which means that ψ must eventually become true and φ must be true up until that point.

There is an efficient algorithm for determining whether a CTL property is true for a circuit that makes use of the symbolic representation of circuits and sets of states described above [6]. For our purposes, the important property of this algorithm is that the of the basic step is performing computations of the following form:

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

(The notation above indicates a series of nested existential quantifications, one for each variable in V.) This expression is called a relational product. Computing relational products is also important for other sequential verification techniques, such as reachability analysis, that use symbolic representations. Thus, it is crucial to be able to do this computation efficiently. A special algorithm is typically used to do this operation in one pass over the BDDs $S(V)$ and $N(V,V')$. By using such an algorithm, it is possible to avoid 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. Up to this point, being forced to construct this BDD has been the major stumbling block in trying to verify complex circuits. In the following sections, we describe how to overcome this problem by using a partitioned transition relation to represent N.

3 Deriving transition relations

The first step in verifying a circuit is to derive its transition relation. For a synchronous circuit with n state variables, we let $V=\{v_0,\ldots,v_{n-1}\}$ and $V'=\{v'_0,\ldots,v'_{n-1}\}$. For each state variable v_i, there is a piece of combinational logic which determines how it is updated. Let f_i be the function computed by this logic. Then v_i's value in the next state is given by

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

These equations are used to define the relations

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

In a legal transition of the circuit, each N_i must be true; hence the transition relation for the circuit is

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

The main point is that the transition relation for a synchronous circuit can be expressed as a conjunction of relations.

In practice, each N_i can often by represented by a small BDD (typically fewer than 100 nodes). However, the size of the BDD representing the entire transition relation may grow as the product of the sizes of the individual parts, and thus may be prohibitively large. In the past, this has been the major limitation of symbolic model checking. For our new method, we instead represent the transition relation by a list of the parts, which are implicitly conjuncted. We call this representation a partitioned transition relation.

4 Computing relational products

As noted earlier, computing relational products is a fundamental operation in many symbolic verification methods. This section describes how relational products can be computed using partitioned transition relations. These techniques significantly increase the size of circuits that can be verified compared to previous methods.

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

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

The main difficulty in computing $S'(V')$ without building the conjunction is that conjunction does not commute with existential quantification. None the less, the new method given below allows the relational product to be computed without constructing the BDD for the full transition relation.

Our new technique is based on two observations. First, circuits exhibit locality, so many of the $N_i(V,V')$ will depend on only a small number of the variables in V and V'. Second, although conjunction does not commute with existential quantification, subformulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. We will take advantage of these observations by conjuncting the $N_i(V,V')$ with $S(V)$ one at a time and quantifying out each variable v when none of the remaining $N_i(V,V')$ depend on v. More formally, the user must chose a permutation ρ of $\{0,\ldots,n-1\}$. This permutation determines the order in which the $N_i(V,V')$ are conjuncted together. For each i, we let D_i be the set of variables in V that $N_i(V,V')$ depends on. Also, let

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

Thus, E_i is the set of variables contained in $D_{\rho(i)}$ that are not contained in $D_{\rho(k)}$ for any k larger than i. The E_i are pairwise disjoint and their union is equal to V. Then the relational product in equation 1 can be computed as

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

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

In the previous section, we described how a sequential circuit could be represented by a set of $N_i(V,V')$, each depending on exactly one variable in V'. While this is almost always more efficient than constructing the full transition relation, it may not be the best choice. As long as the BDDs do not get too large, it is better to combine several of the $N_i(V,V')$ into one BDD (by taking their conjunction) in order to reduce overhead.

5 A synchronous pipeline

In this section, we discuss the verification of a simple pipeline circuit to illustrate the efficiency of the techniques discussed in the previous section. This circuit, shown in figure 1 and first described in an earlier paper [6], performs three-address arithmetic and logical operations on operands stored in a register file.

The pipeline decomposes naturally into pieces such as general registers, operand registers, etc. We used this decomposition as a starting point for breaking the transition relation into parts. Some of the parts, such as the register file, were found to require large BDDs to represent; we broke these into more pieces. We also found that we could combine some of the parts, such as the individual pipe registers, without increasing the number of BDD nodes required; we did this to decrease overhead. The final decomposition had the following pieces: control logic; pipe registers; the first ALU operand register; the second ALU operand register; and one piece for each general register. This ordering was also the ordering ρ used for processing the transition relation. With this ordering, the number of variables in intermediate results never exceeded the number of state variables by more than the number of bits in a register. We also found that the sizes of the intermediate results with this ordering increased monotonically during each step. In other words, breaking the transition relation into pieces did not result in having to manipulate larger BDDs than would have been necessary with a monolithic transition relation. This is an important

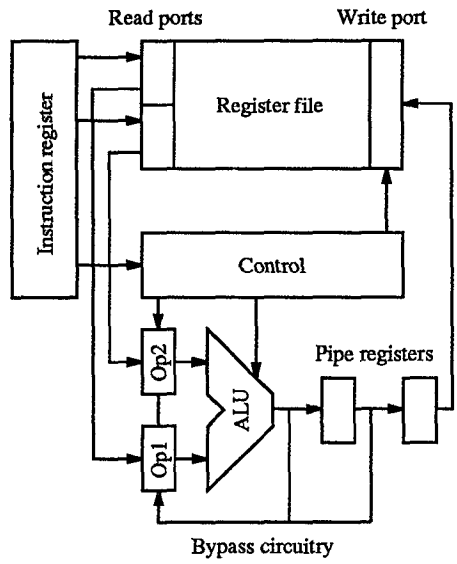

Figure 1: Pipeline circuit block diagram

point; in many applications involving BDDs, it is the number of nodes in intermediate results, not the final result, that limits the size of the problem that can be handled.

In the BDD variable ordering that we used, the source address registers are closest to the root. The bits of these registers are interleaved. These are followed by variables which make up the destination address shift chain. For each stage in the chain, starting with the leftmost (input) stage, there is a stall bit followed by a destination address register. Next come the opcode shift registers, with the bits interleaved. The operand registers, general registers and pipe registers, interleaved and arranged most significant bit to least significant bit, are at the end of the ordering.

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

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

The $\log r$ factors arise because an extra addressing bit is needed when r increases from 2^i to $2^{i+1} - 1$. The number of parts in the transition relation increased linearly with r, and did not depend on w, s or o. The number of BDD nodes in each piece of the transition relation was typically between 10 and 500. No piece ever had more than 1,500 nodes. The

way the sizes of the pipe registers and ALU operand registers vary with o depends on the exact operations. The ones we used were addition, subtraction, and bitwise logical operations. With this set, the control logic grew $O(\log o)$, the pipe registers and ALU operand registers grew $O(o)$, and the general registers did not vary with o.

We also studied the BDDs representing the various state sets in the verification and used profiles to determine their rates of growth. Since most of the time and space for each verification was used computing and representing the value of the destination register at the end of the current operation, we concentrated on these. The number of nodes in these BDDs grows as $O\big(rs(r+s)\log r + w(r+s)^2\big)$. More intricate analysis can be used to show, for the operations above, that the sizes are also bounded by $O(o^2)$. The largest BDDs we encountered had slightly less than 12,500 nodes; typical sizes were about 1,000 nodes.

We performed the tests described above using a CTL model checker written mostly in the T dialect of LISP [13]. The actual BDD manipulation routines are written in C and are based on a package by Brace, Rudell and Bryant [3]. The model checker was run on a Sun 4. Figure 2 shows how the verification time depends on the parameters r, w, s and o. The following table shows the values used for the fixed parameters in these tests.

	r	w	s	o
vary r		1	1	1
vary w	3		1	1
vary s	2	2		1
vary o	2	3	1	

The figure can be used to estimate the asymptotic increase in verification time as a function of the different parameters. Because the figure is a log-log plot, a straight line with slope m indicates that the verification time grows as the mth power of the independent variable. We used linear regression to determine the approximate rate at which the verification times increased. We can divide the graph for r into groups of points where r increases from 2^i to $2^{i+1}-1$ for some i. The slopes of the best fit lines within the last two groups of points are 2.48 and 2.50. The verification time increases significantly when r increases to 2^{i+1} because each of the addresses requires an extra bit. In the graphs for w, s and o, the slopes of the best fit lines through the last half of the points are 2.16, 1.92 and 1.54, respectively.

It is important to note that in all cases, the verification time is growing polynomially in the number of *components* of the system. Polynomial verification times were also documented in earlier work [6, 7]. Other researchers [1] using symbolic techniques have demonstrated verification times that grow sublinearly in the number of *states* of the system, but still exponentially in the number of components.

For comparison, we also ran the verification with a monolithic transition relation. With 8 bit registers, the monolithic transition relation required more than 75,000 BDD nodes to represent, compared with less than 750 nodes using a partitioned transition relation, a difference of more than two orders of magnitude. In addition, the verification needed nearly an order of magnitude more time. We also note that combining parts of a transition relation can result in higher

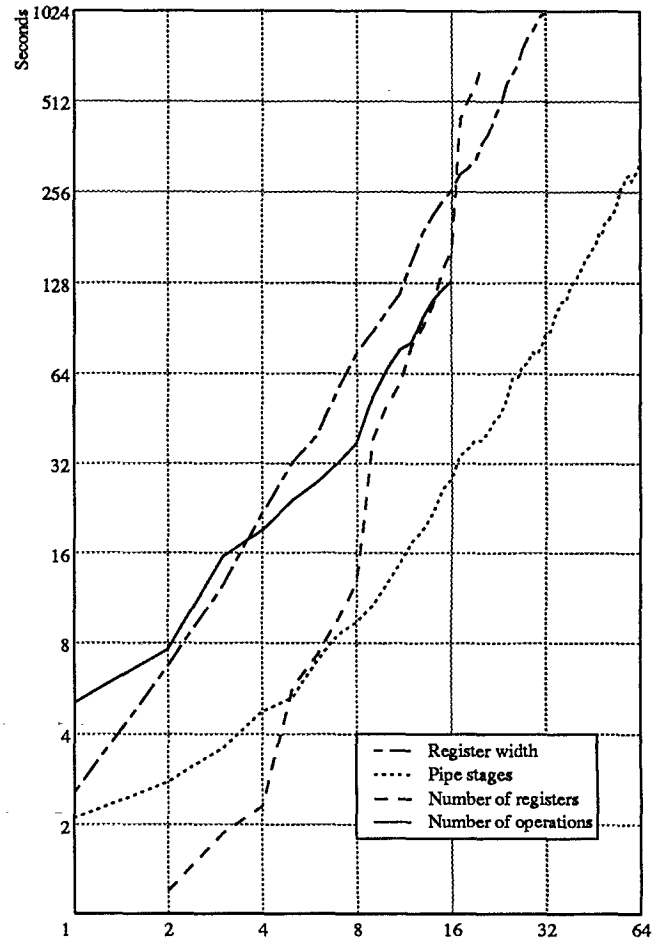

Figure 2: Verification times

asymptotic complexity. For example, the total number of nodes in the BDDs that represent the register file in the partitioned transition relation is $O(r)$, while the BDD for their conjunction has $O(r^2)$ nodes.

We also ran several more realistic examples. The largest of these was a pipeline with 8 registers, each 32 bits wide, 2 pipe registers, and one operation. This example had 406 state variables resulting in more than 10^{120} reachable states, and the verification took 4 hours and 20 minutes of CPU time.

6 Discussion and future research

By using partitioned transition relations, we have significantly improved the efficiency of symbolic model checking. For some of the examples discussed in section 5, we were able to reduce the verification time by more than an order of magnitude and the number of nodes needed to represent the transition relation by two orders of magnitude. We verified a circuit with more than 400 state variables and over 10^{120} reachable states.

For deterministic systems, a transition function vector can be used to represent how a circuit transitions from one state

to another. In this method, a separate BDD is used for each state holding node of the system. This BDD represents the function computed by the combinational logic driving the associated node. Coudert *et al.* [9, 10] describe a number of algorithms for manipulating transition functions. They note that the monolithic transition relation can require many more BDD nodes than the corresponding transition function vector [10]. However, they report that computations with transition relations are faster than those using transition functions. Partitioned transition relations provide the speed of transition relations and the memory efficiency of transition functions.

Recently, Touati *et al.* [14] proposed another method for representing transition relations as implicit conjunctions. They use the *constrain* operator of Coudert *et al.* [9] to eliminate the state set $S(V)$ in equation 1. Then they compute the resulting conjunction as a balanced binary tree, quantifying out each variable in V when all the BDDs depending on that variable have been combined. We believe that this method is inferior to the one proposed here because the constrain operator may introduce dependencies on any of the variables in $S(V)$. This makes it impossible to compute in advance a schedule for quantifying out the variables in V, which in turn turn reduces the practicality of caching results between relational product computations. In addition, if $S(V)$ depends on most of the variables in V, it may not be possible to quantify out many variables before performing the final conjunction. They also suggest having one transition relation per state variable. In our experience, it is often better to combine parts of the transition relations to reduce overhead; this idea is also applicable to their method. We implemented their method and tested it on some of the examples in section 5. For a pipeline with four 8 bit registers, one pipe register and one operation, our method was more than five times faster. In addition, for some of the relational product computations, the intermediate BDDs using their method were more than an order of magnitude larger than the final result.

There are several questions that would benefit from future research. One problem is finding automatic methods for determining efficient orders in which to process parts of the transition relation. Some questions still remain concerning the exact relationships between using transition relations and using transition functions, and between model checking and state machine comparison. Using transition functions and state machine comparison, Coudert *et al.* have observed verification times which were sublinear in the number of states in the system, but these times were still exponential in the number of components [1]. In the examples we have considered, using model checking and transition relations, the verification time has grown polynomially in the number of components. When we tried to apply state machine comparison to the pipeline given in the previous section, we found that the size of the BDD representing the reachable state set grew exponentially with the number of registers and the number of pipe stages. It would be interesting to try to categorize some of the circuit features that influence the complexity of the different methods of verification. We believe that the kind of asymptotic analysis done in the previous section, when applied to a larger class of circuits and verification techniques, would be useful for this purpose.

References

[1] C. Berthet, O. Coudert, and J.-C. Madre. New ideas on symbolic manipulations of finite state machines. In *ICCD*, 1990.

[2] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *ICCD*, 1989.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC*, 1990.

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

[5] R. E. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In Kurshan and Clarke [11].

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *DAC*, 1990.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[8] 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), 1986.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer-Verlag, 1989.

[10] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Kurshan and Clarke [11].

[11] R. Kurshan and E. M. Clarke, editors. *Workshop on Computer-Aided Verification*. DIMACS, June 1990. Technical Report 90-31.

[12] K. L. McMillan. Formal verification of the Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, 1991. To appear.

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

[14] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *ICCAD*, 1990.