Model Checking and Abstraction*

Edmund M. Clarke
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
emc+@cs.cmu.edu

Orna Grumberg
Computer Science Department
The Technion
Haifa 32000, Israel
orna@cs.technion.ac.il

David E. Long
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
long+@cs.cmu.edu

October 16, 1991

Abstract

We describe a method for using abstraction to reduce the complexity of temporal logic model checking. The basis of this method is a way of constructing an abstract model of a program without ever examining the corresponding unabstracted model. We show how this abstract model can be used to verify properties of the original program. We have implemented a system based on these techniques, and we demonstrate their practicality using a number of examples, including a pipelined ALU circuit with over 10^{1300} states.

1 Introduction

Complicated finite state programs arise in many applications of computing—particularly in the design of hardware controllers and communication protocols. When the number of states is large, it may be very difficult to determine if such a program is correct. Temporal logic model checking [5, 15, 16, 17] is a method for automatically deciding if a finite state program satisfies its specification. A model checking algorithm for the propositional branching time temporal logic CTL was presented at the 1983 POPL conference [6]. The algorithm was linear in both the size of the transition system (or model) determined by the program and in the length of its specification. In the paper, it was used to verify a simple version of the alternating bit protocol with 20 states.

In the nine years that have passed since that paper was published, the size of the programs that can be verified by this means has increased dramatically. By developing special programming languages for describing transition systems, it became possible to check examples with several thousand states. This was sufficient to find subtle errors in a number of nontrivial, although relatively small, protocols and circuit designs [1]. Use of boolean decision diagrams (BDDs) [2] led to an even greater increase in size. Representing transition relations implicitly using BDDs made it possible to verify examples that would have required 10^{20} states with the original version algorithm [4]. Refinements of the BDD-based techniques [3] have pushed the state count up over 10^{100} states. In this paper, we show that by combining model checking with abstraction, we are able to handle even larger systems. In one example, we are able to verify a pipelined ALU circuit with 64 registers, each 64 bits wide, and more than 10^{1300} reachable states.

Our paper consists of three main parts. In the first, we propose a method for obtaining abstract models of a program. In the second, we show how these abstract models can be used to verify properties of the program. Finally, we suggest a number of useful abstractions, and we illustrate them via a series of examples.

We model programs as transition systems in which the states are n-tuples of values. Each component of a state represents the value of some variable. If the ith component ranges over the set D_i, then the set of all program states is $D_1 \times \cdots \times D_n$. Abstractions will be formed by giving surjections $h_1, \ldots, h_n$ which map each D_i onto a set D_i' of abstract values. The surjection $h = (h_1, \ldots, h_n)$ then maps each program state to a corresponding abstract state. This mapping may be applied in a natural way to the initial states and the transitions of the program. The result is a transition system which we refer to as the *canonical abstraction*

*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 No. CCR-9005992 and the U.S.-Israeli Binational Science Foundation.

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 U.S. government.

of the original program. If it is possible to construct this abstraction, we can use it to verify properties of the program. However, if the state space of the transition system is infinite or very large, this may not be feasible. In the finite state case, it may be possible to represent the system using BDD-based methods, but the computational complexity of building the canonical abstraction may still be very high. To circumvent these problems, we show how to derive an *approximation* to the canonical abstraction. The approximation may be constructed directly from the text of the program without first building the original transition system. We show how this can be accomplished by symbolic execution of the program over the abstract state space.

The specification language that we use is a propositional temporal logic called CTL* [7]. This logic combines both branching time operators and linear time operators and is very expressive. Formulas are formed using the standard operators of linear temporal logic and two path quantifiers, $\forall$ and $\exists$. The formula $\forall(\phi)$ is true at a state whenever ϕ holds on all computation paths starting at the state. The formula $\exists(\phi)$ is true whenever ϕ holds for some computation path. The atomic state formulas in the logic are used to specify that a program variable has a particular abstract value. Because of this, formulas of the logic may be interpreted with respect to either the original transition system or its abstraction. Our goal is to check the truth value of a formula in the abstract system, and conclude that it has the same truth value in the original system. We prove that this approach is *conservative* if we restrict to a subset of the logic called $\forall$CTL* [12] in which only the $\forall$ path quantifier is allowed. If a formula is true in the abstract system, we can conclude that the formula is also true in the original system. However, if a formula is false in the abstract system, it may or may not be false in the original system. In addition, we show that if the equivalence relations induced by the h_i are congruences with respect to the operations used in the program, then the method is *exact* for full CTL*. That is, a formula is true in the abstract system if and only if it is true of the original system.

We suggest several different abstractions that are useful for reasoning about programs. These abstractions include

1. congruence modulo an integer, for dealing with arithmetic operations;
2. single bit abstractions, for dealing with bitwise logical operations;
3. product abstractions, for combining abstractions such as the above; and
4. symbolic abstractions. This is a powerful type of abstraction that allows us to verify an entire class of formulas simultaneously.

We demonstrate the practicality of our methods by considering a number of examples, some of which are too complex to be handled by the BDD-based methods alone. These examples include a 16 bit by 16 bit hardware multiplier and a pipelined ALU circuit with over 4000 state variables.

Numerous other authors have considered the problem of reducing the complexity of verification by using equivalences, preorders, etc. For example, Graf and Steffen [11] describe a method for generating a reduced version of the global state space given a description of how the system is structured and specifications of how the components interact. Clarke, Long and McMillan [8] describe a related attempt. Grumberg and Long [12] propose a framework for compositional verification based on $\forall$CTL*. Dill [10] has developed a trace theory for compositional design of asynchronous circuit. But, these methods are mainly useful for abstracting away details of the control part of a system.

There has been relatively little work on applying model checking to systems which manipulate data in a nontrivial way. Wolper [18] demonstrates how to do model checking for programs which are data independent. This class of programs, however, is fairly small. Our approach makes it possible to handle programs which have some data dependent behavior. More recently, BDD-based model checking techniques [4, 9] have been used to handle circuits with data paths. These methods, while much more powerful than explicit state enumeration, are still unable to deal with some systems of realistic complexity. Some examples in section 9, for instance, could not be handled directly with these approaches. Our method works well in conjunction with these techniques, however.

Of the work on using abstraction to verify finite state systems, the approach described by Kurshan [14] is most closely related to ours. This approach has been automated in the COSPAN system [13]. The basic notion of correctness is ω-language containment. The user may give abstract models of the system and specification in order to reduce the complexity of the test for containment. To ensure soundness, the user specifies homomorphisms between the actual and abstract processes. These homomorphisms are checked automatically. Our work differs from Kurshan's in several important respects.

1. Our specifications are given in the temporal logic CTL* which can express both branching time and linear time properties. Moreover, we are able to identify precisely a large class of temporal formulas for which our verification methodology is sound. Not all properties are preserved in going from the reduced system to the original, so this is quite important.
2. Our abstractions correspond to language homo-

morphisms induced by boolean algebra homomorphisms in Kurshan's work. For this type of abstraction, we show how to derive automatically an *approximation* to the abstracted state machine. This approximation is constructed directly from the program, so that *it is unnecessary to examine the state space of the unabstracted machine.* There is no need to check for a homomorphism between the abstract and unabstracted systems, and it is possible to apply our technique to construct approximations for systems with infinite state spaces.

3. The particular abstraction mappings that we use also appear to be new. We demonstrate that these abstractions are powerful enough and that the corresponding approximations are accurate enough to allow us to verify interesting properties of complex systems.

Our paper is organized as follows: the next section is a brief introduction to BDDs and symbolic model checking. This is followed by a discussion of transition systems, homomorphisms, and the notion of abstraction that we use. Section 4 discusses the compilation of programs into transition systems. In the following section, we show how to construct the approximation directly from a program without first building the original transition system. The conditions required for exactness are discussed in section 6. Section 7 is the heart of our paper; we relate the theory developed in the previous sections to the temporal logic that we use for specifications. In particular, we prove that our method is conservative in the case of ∀CTL* formulas. We also show that if the approximation is exact, then all CTL* formulas are preserved. Section 8 describes the programming language that is used for specifying finite state systems. Section 9 explains some of the abstractions that we have developed for reasoning about complex systems and illustrates their use with examples. The paper concludes with a discussion of some directions for future research.

2 Boolean decision diagrams

Boolean decision diagrams (BDDs) are a canonical form representation for boolean formulas described by Bryant [2]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. A BDD is similar to a boolean 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

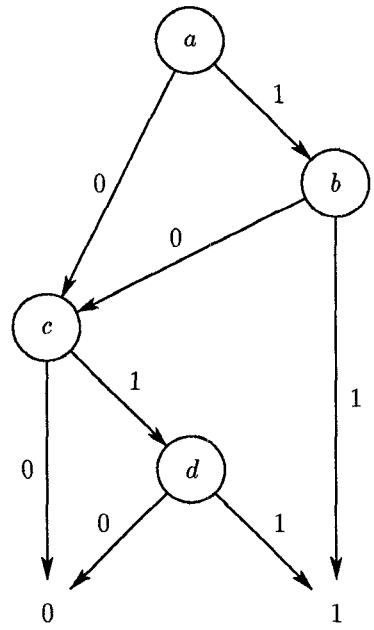


Figure 1: A BDD representing $(a \wedge b) \vee (c \wedge d)$

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 valuation $\{ a = 1, b = 0, c = 1, d = 1 \}$ leads to a leaf node labeled 1, hence the formula is true for this assignment.

Bryant showed that given a variable ordering, there is a canonical BDD for every formula. He also gives algorithms of linear complexity for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for formulas f and g. Quantification over boolean variables and substitution of a variable by a formula are also straightforward using this representation.

Given a finite state program, let V be its set of boolean state variables. 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 program; hence the formula may be thought of as representing a set of program states. The BDD for the formula is in practice a concise representation for this set of states. In addition to representing sets of states of a program, we must represent the transitions that the program 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 of the program. Such a pair can be viewed as corresponding to a transition between the states of the pair. Thus, we can represent sets of transitions using BDDs in much the same way as we represent sets of states. Many verification algorithms such as temporal logic model checking and state machine comparison can make effective use of this representation.

3 Transition systems and abstractions

We consider programs with a finite set of variables $v_1, v_2, \ldots, v_n$. If each variable v_i ranges over a set D_i of possible values, then the set of all possible program states is $D_1 \times D_2 \times \cdots \times D_n$, which we denote by D. We represent the possible behaviors of the program with a set of transitions between states. This notion is formalized in the following definition.

Definition 1 *A transition system over D is a triple $M = \langle S, I, R \rangle$ where*

1. *$S = D$ is a set of states;*
2. *$I \subseteq S$ is a set of initial states; and*
3. *$R \subseteq S \times S$ is a transition relation.*

Abstractions will be formed by letting the program variables range over sets D'_i of abstract values. We will give mappings to specify the correspondence between unabstracted and abstracted values. Formally, we let $h_1, h_2, \ldots, h_n$ be surjections, with $h_i : D_i \to D'_i$ for each i. These mappings induce a surjection $h : D \to D'$ defined by

$$h\big((d_1, \ldots, d_n)\big) = \big(h_1(d_1), \ldots, h_n(d_n)\big).$$

Alternatively, the relation between unabstracted and abstracted values can be specified by means of a set of equivalence relations. In particular, each h_i corresponds to the equivalence relation $\sim_i \subseteq D_i \times D_i$ defined by

$$d_i \sim_i e_i \qquad \text{if and only if} \qquad h_i(d_i) = h_i(e_i).$$

The mapping h induces an equivalence relation $\sim \subseteq D \times D$ in the same manner: $(d_1, \ldots, d_n) \sim (e_1, \ldots, e_n)$ if and only if $d_1 \sim_1 e_1 \wedge \cdots \wedge d_n \sim_n e_n$. We will sometimes specify abstractions by mappings and sometimes specify them by equivalence relations. The two methods are entirely equivalent.

Fix a transition system M over D and a surjection $h : D \to D'$. By applying h to the components of M, we obtain an abstract version of M.

Definition 2 *The canonical abstraction of M induced by h is the transition system M_{abs} over D' defined as follows.*

1. *$S_{\text{abs}} = D'$.*
2. *$I_{\text{abs}}(d')$ if and only if $\exists d \big(h(d) = d' \wedge I(d)\big)$.*
3. *$R_{\text{abs}}(d', e')$ if and only if*

$$\exists d \exists e \big(h(d) = d' \wedge h(e) = e' \wedge R(d, e)\big).$$

Definition 3 *A homomorphism from a transition system M over D to a transition system M' over D' is a surjection $h : D \to D'$ such that:*

1. *$I(d)$ implies $I'\big(h(d)\big)$; and*
2. *$R(d, e)$ implies $R'\big(h(d), h(e)\big)$.*

Proposition 1 *The mapping h from M to M_{abs} is a homomorphism.*

As we will show in section 7, an abstract transition system such as M_{abs} may be used to deduce properties of M. Moreover, using an abstract transition system instead of M may greatly reduce the complexity of automatically verifying these properties. Unfortunately, it is often expensive or impossible to construct M_{abs} directly because we must have a representation of M to do the abstraction. We may not be able to obtain such a representation if D is infinite or simply too large for our system to handle. In BDD-based systems, even if we are able to represent M, the complexity of computing the relational products in the definition of M_{abs} is often extremely high. In section 5, we discuss a method for circumventing these problems. The basic idea will be to take advantage of structure in the transition system M. Such structure arises because M is typically given by a relatively concise program. We show how to compute an approximation to M_{abs} that can be derived directly from the program text. Hence, it is never necessary to construct a representation of M. In addition, the approximation is often accurate enough to allow us to verify interesting properties of the program.

4 Compilation

The approximation to M_{abs} will be constructed by performing an "abstract compilation" of the program. Hence we begin by considering how programs are compiled into transition systems. At a conceptual level, the compilation may be viewed as a two step process. First, predicate logic formulas $\mathcal{R}$ and $\mathcal{I}$ are constructed to represent the program's actions and initial states. These formulas are built from a set of primitive relations that represent the operations such as addition and comparison used in the program. Second, the formulas are interpreted to derive the actual transition system. By thinking of the process at this level, we can avoid low-level details which would tie the discussion to a particular programming language. Below, we illustrate how formulas representing the actions and initial states might be derived. The construction is similar to that used to give the relational semantics of an imperative programming language or to derive verification conditions in the inductive assertions method.

First note that a precondition-postcondition semantics is not sufficient for our purposes since we are interested in the temporal behavior of programs. For this reason, there must be some convention about when time passes during the execution of the program. We assume that there are a finite set of control points in the program (typically chosen by the user) and that executing a sequence of statements between two consecutive control points requires exactly one time unit. To avoid having infinite sequences of statements take a finite amount of time, we assume that every path through a loop body

must contain at least one control point. Hence it is not necessary to deal with loops explicitly; they are implicit in the sequencing of transitions between control points. We will assume that the intervals between consecutive control points are sequences of assignment statements and boolean conditions that are derived from the conditional statements in the program. To begin, we examine how formulas representing these intervals can be derived.

Consider an assignment statement, say $v_i := v_j$. The formula $T(v_1, \ldots, v_n, v_1', \ldots, v_n')$ for such a statement contains two parts. The first specifies that the value of v_i after the statement (referred to as v_i') is equal to the value of v_j before the statement. The second specifies that no other variables change. For this particular example, we obtain

$$v_i' = v_j \wedge \bigwedge_{k \neq i} v_k' = v_k.$$

When the statement is more complex, we introduce additional temporary variables to hold intermediate results. For example, $v_i := v_i + (v_j - v_k)$ would be represented by the formula

$$\exists t \left(P_-(v_j, v_k, t) \wedge P_+(v_i, t, v_i')\right) \wedge \bigwedge_{l \neq i} v_l' = v_l,$$

where P_- and P_+ are primitive relations representing subtraction and addition, respectively.

For boolean conditions, such as $v_i > v_j$, T again contains two parts. The first specifies that the condition evaluates to the boolean value *true* (represented by the primitive relation P_{true}). The second specifies that no variables change. For this example, we obtain

$$\exists t \left(P_{true}(t) \wedge P_>(v_i, v_j, t)\right) \wedge \bigwedge_{k} v_k' = v_k.$$

To find the formula representing a sequence S_1; S_2, we first find the formulas T_1 representing S_1 and T_2 representing S_2. The formula for the sequence is formed by taking a relational product:

$$\exists v_1'' \ldots \exists v_n'' \left(T_1(v_1, \ldots, v_n, v_1'', \ldots, v_n'') \wedge T_2(v_1'', \ldots, v_n'', v_1', \ldots, v_n')\right).$$

The logical formula for the entire program is formed by combining the formulas for its intervals. We introduce an additional program variable p that ranges over the set of the program's control points. We also assume that there are primitive relations representing each of the individual control points. The formula for the entire program is a disjunction with one disjunct per interval. Each of these disjuncts is of the form

$$C_j(p) \wedge T_{jk}(v_1, \ldots, v_n, v_1', \ldots, v_n') \wedge C_k(p'),$$

where C_j is the relation for control point j, C_k is the relation for control point k, and T_{jk} is the formula for the interval between control points j and k.

In an actual implementation, it is possible to avoid enumerating all intervals by treating the program graph as a DAG rather than a tree. In addition, formulas representing the transition relation and initial states of the program will not actually be constructed; instead, the program is "symbolically executed" to derive the corresponding transition system. Starting from the initial states of the program, we simulate the execution of the program from each state. As we do the simulation, we record which states transitioned to which other states. A key point is that this simulation is driven by knowing how the operations in the program behave, i.e., how the primitive relations are interpreted.

5 Computing approximations

In the previous section, we mentioned that the initial states and transition relation of a transition system M could be represented by formulas $\mathcal{I}$ and $\mathcal{R}$. Now we examine the relationship between these formulas and similar formulas $\mathcal{I}_{\text{abs}}$ and $\mathcal{R}_{\text{abs}}$ for M_{abs}. By applying a certain transformation to these latter formulas, we obtain formulas $\mathcal{I}_{\text{app}}$ and $\mathcal{R}_{\text{app}}$ describing an approximation M_{app} to M_{abs}. Throughout this section and the next, we assume that ϕ and ψ are relations built up from the primitive relations representing the operations in the program.

Recall that building M_{abs} requires evaluating two relational products, both involving existential quantification over the elements of D. For conciseness, we will denote this kind of existential abstraction using an operator $[\cdot]$. That is $[\phi(x_1, \ldots, x_m)]$ is an abbreviation for

$$\exists y_1 \ldots \exists y_m \left(h(y_1) = x_1 \wedge \cdots \wedge h(y_m) = x_m \wedge \phi(y_1, \ldots, y_m)\right).$$

Note that $[\phi(x_1, \ldots, x_m)]$ has the same free variables as $\phi(x_1, \ldots, x_m)$. In the latter, the variables range over elements in the D_i, while in the former they range over elements in the D_i'. Based on the definition of M_{abs}, we observe that if $\mathcal{I}$ and $\mathcal{R}$ are the formulas representing I and R, then $\mathcal{I}_{\text{abs}} = [\mathcal{I}]$ and $\mathcal{R}_{\text{abs}} = [\mathcal{R}]$ are formulas representing I_{abs} and R_{abs}.

We now define a transformation $\mathcal{T}$ on formulas $[\phi]$. The idea of $\mathcal{T}$ will be to simplify the formulas to which $[\cdot]$ is applied. We assume that ϕ is given in negation normal form, i.e., negations are applied only to primitive relations.

1. If P is a primitive relation
$$\mathcal{T}([P(x_1, \ldots, x_m)]) = [P(x_1, \ldots, x_m)]$$
$$\mathcal{T}([\neg P(x_1, \ldots, x_m)]) = [\neg P(x_1, \ldots, x_m)]$$
2. $\mathcal{T}([\phi \wedge \psi]) = \mathcal{T}([\phi]) \wedge \mathcal{T}([\psi])$.
3. $\mathcal{T}([\phi \vee \psi]) = \mathcal{T}([\phi]) \vee \mathcal{T}([\psi])$.

4. $\mathcal{T}([\exists x\, \phi]) = \exists x\, \mathcal{T}([\phi])$.

5. $\mathcal{T}([\forall x\, \phi]) = \forall x\, \mathcal{T}([\phi])$.

In other words, $\mathcal{T}$ pushes the existential abstractions inwards.

We note that applying $\mathcal{T}$ to a formula results in a formula which is true more often. This is an important point since, as will be seen in section 7, it ensures that our methodology will be conservative. Formally, we have the following result.

Theorem 1 $[\phi] \rightarrow \mathcal{T}([\phi])$.

We will let M_{app} be the transition system over D' whose initial states and transition relation are represented by the formulas $\mathcal{I}_{\text{app}} = \mathcal{T}(\mathcal{I}_{\text{abs}})$ and $\mathcal{R}_{\text{app}} = \mathcal{T}(\mathcal{R}_{\text{abs}})$ respectively.

Proposition 2 *The following relationships hold between the components of M_{abs} and the components of M_{app}.*

1. $S_{\text{abs}} = S_{\text{app}}$;
2. $I_{\text{abs}} \subseteq I_{\text{app}}$; *and*
3. $R_{\text{abs}} \subseteq R_{\text{app}}$.

Observe that $\mathcal{I}_{\text{app}}$ and $\mathcal{R}_{\text{app}}$ have essentially the same structure as $\mathcal{I}$ and $\mathcal{R}$. The only difference is at the lowest level; the latter formulas have primitive relations and their negations, while the former have *abstract* versions of these same relations. Thus, just as we can derive M by symbolically executing the program using the interpretations of the primitive relations, we can derive M_{app} by symbolically executing the program using the abstracted interpretations of the primitive relations.

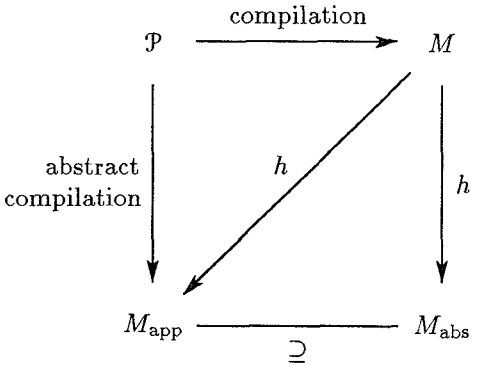
We now consider the relation between M and M_{app}. Recall that the mapping h is a homomorphism from M to M_{abs}. We also have the following property of homomorphisms.

Proposition 3 *Let h be a homomorphism from M to M', and let M'' be a transition system such that*

1. $S' = S''$;
2. $I' \subseteq I''$; *and*
3. $R' \subseteq R''$.

Then h is a homomorphism from M to M''.

Using the above properties, we can conclude that h is also a homomorphism from M to M_{app}. We will discuss the properties implied by this fact in section 7. The relationship between M, M_{abs} and M_{app} is summarized by the following diagram; here $\mathcal{P}$ is a program.



6 Exact approximations

In the previous sections, we demonstrated the existence of homomorphisms from M to M_{abs} and M_{app}. These results will be used to show that our verification methodology is conservative. In this section, we consider additional properties which suffice to make the method exact. Recall that each h_i induces an equivalence relation $\sim_i$ on D_i.

Definition 4 *Let $P(x_1, \ldots, x_m)$ be a relation with x_j ranging over D_{i_j}. The equivalence relations $\sim_{i_j}$ are a congruence with respect to P if*

$$\forall d_1 e_1 \ldots d_m e_m \, (d_1 \sim_{i_1} e_1 \wedge \cdots \wedge d_m \sim_{i_m} e_m \rightarrow (P(d_1, \ldots, d_m) \Leftrightarrow P(e_1, \ldots, e_m))).$$

Theorem 2 *If the $\sim_i$ are congruences with respect to the primitive relations and $\psi = \mathcal{T}^*([\phi])$, then $\psi \Leftrightarrow [\phi]$.*

Corollary 1 *If the $\sim_i$ are congruences with respect to the primitive relations, then $M_{\text{abs}} = M_{\text{app}}$.*

Definition 5 *An exact homomorphism from a transition system M to a transition system M' is a homomorphism h from M to M' with the following additional properties.*

1. $I'(h(d))$ *implies* $I(d)$; *and*
2. $R'(h(d), h(e))$ *implies* $R(d, e)$.

Theorem 3 *If the $\sim_i$ are congruences with respect to the primitive relations, then h is an exact homomorphism from M to M_{abs} (and hence to M_{app}).*

7 Temporal logic

The logics that we will use for specifying properties will be subsets of the logic CTL*. CTL* is a powerful temporal logic that can express both branching time and linear time properties. For convenience when defining subsets of the logic, we will assume that all formulas are given in negation normal form. That is, negations only appear in atomic state formulas.

Definition 6 *The logic CTL* [7] is the set of state formulas given by the following inductive definition.*

1. *true and false are atomic state formulas. If v_i is a program variable and $d'_i \in D'_i$, then $v_i \equiv d'_i$ and $v_i \not\equiv d'_i$ are atomic state formulas.*
2. *If ϕ and ψ are state formulas, then $\phi \wedge \psi$ and $\phi \vee \psi$ are state formulas.*
3. *If ϕ is a path formula, then $\forall(\phi)$ and $\exists(\phi)$ are state formulas.*
4. *If ϕ is a state formula, then ϕ is a path formula.*
5. *If ϕ and ψ are path formulas, then so are $\phi \wedge \psi$ and $\phi \vee \psi$.*
6. *If ϕ and ψ are path formulas, then so are $\mathbf{X}\phi$, $\phi\,\mathbf{U}\,\psi$, and $\phi\,\mathbf{V}\,\psi$.*

We also use the following abbreviations: $\mathbf{F}\phi$ and $\mathbf{G}\phi$, where ϕ is a path formula, denote $(true\mathbf{U}\phi)$ and $(false\mathbf{V}\phi)$ respectively.

CTL is the subset of CTL* that is obtained by eliminating rules 3 through 6 above and adding the rule.

3'. If ϕ and ψ are state formulas, then so are $\forall\mathbf{X}\phi$, $\exists\mathbf{X}\phi$, $\forall(\phi\,\mathbf{U}\,\psi)$, $\exists(\phi\,\mathbf{U}\,\psi)$, $\forall(\phi\,\mathbf{V}\,\psi)$, and $\exists(\phi\,\mathbf{V}\,\psi)$.

CTL is of interest because there is a very efficient model checking algorithm for it [7]. ∀CTL* and ∀CTL [12] are restricted subsets of CTL* and CTL respectively in which the only path quantifier allowed is ∀. These two logics are sufficient to express many of the properties that arise when verifying programs. As we will see, these logics will also be used when the conditions needed for exactness do not hold.

We now define the semantics of CTL* for transition systems M over D or D'. The atomic state formulas will be interpreted slightly different depending on whether the state set is abstract or not.

Definition 7 *A path in M is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for every $i \in \mathbb{N}$, $R(s_i, s_{i+1})$.*

The notation π^n will denote the suffix of π which begins at s_n.

Definition 8 *Satisfaction of a state formula ϕ by a state s ($s \models \phi$) and of a path formula ψ by a path π ($\pi \models \psi$) is defined inductively as follows.*

1. *$s \models true$, and $s \not\models false$. If $s = (e_1, \ldots, e_n) \in D$, then $s \models (v_i \equiv d'_i)$ if and only if $h_i(e_i) = d'_i$. If $s = (e'_1, \ldots, e'_n) \in D'$, then $s \models (v_i \equiv d'_i)$ if and only if $e'_i = d'_i$. In either case, $s \models (v_i \not\equiv d'_i)$ if and only if it is not the case that $s \models (v_i \equiv d'_i)$.*
2. *$s \models \phi \wedge \psi$ if and only if $s \models \phi$ and $s \models \psi$. $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$.*
3. *$s \models \forall(\phi)$ if and only if for every path π starting at s, $\pi \models \phi$. $s \models \exists(\phi)$ if and only if there exists a path π starting at s such that $\pi \models \phi$.*
4. *$\pi \models \phi$, where ϕ is a state formula, if and only if the first state of π satisfies the state formula.*
5. *$\pi \models \phi \wedge \psi$ if and only if $\pi \models \phi$ and $\pi \models \psi$. $\pi \models \phi \vee \psi$ if and only if $\pi \models \phi$ or $\pi \models \psi$.*
6. *$\pi \models \mathbf{X}\phi$ if and only if $\pi^1 \models \phi$. $\pi \models \phi\,\mathbf{U}\,\psi$ if and only if there exists $n \in \mathbb{N}$ such that $\pi^n \models \psi$ and for all $i < n$, $\pi^i \models \phi$. $\pi \models \phi\,\mathbf{V}\,\psi$ if and only if for all $n \in \mathbb{N}$, if $\pi^i \not\models \phi$ for all $i < n$, then $\pi^n \models \psi$.*

The notation $M \models \phi$ indicates that every initial state of M satisfies the formula ϕ.

We now turn to the main theorems. These results tell us when it is sound to use abstraction to verify properties of a program.

Theorem 4 *Suppose h be a homomorphism from M to M' and ϕ is a ∀CTL* formula. Then $M' \models \phi$ implies $M \models \phi$.*

Theorem 5 *Suppose h is an exact homomorphism from M to M' and ϕ is a CTL* formula. Then $M \models \phi$ if and only if $M' \models \phi$.*

8 A simple language

In this section, we briefly describe a language for specifying reactive programs. We will use this language in the examples that follow. The language is procedural and contains structured programming constructs, such as `while` loops and non-recursive procedures. It is also finite state: the user must specify a fixed number of bits for each input and output in a program. The model of computation is a synchronous one. At the start of each time step, inputs to the program are obtained from the environment. All computation in a program is viewed as instantaneous. There is one special statement, `wait`, which is used to indicate the passage of time. When a `wait` statement is encountered, changes to the program's outputs become visible to the environment, and a new time step is initiated. Thus, computation proceeds as follows: obtain inputs, compute until a `wait` is encountered, make output changes visible, obtain new inputs, etc. Aside from the `wait` statement, most of the language features are self-explanatory.

A program in the language may be compiled into a Moore machine for verification. Since the Moore machine for a program may have a large number of states (even after abstraction), it is important not to generate an explicit-state representation of this machine. Instead, our compiler directly produces a BDD that represents the Moore machine. This BDD is used as the input to a BDD-based model checking program. When a program is compiled, the user may specify abstractions for some of the inputs or outputs. By using the techniques described earlier, the compiler directly generates an (approximate) abstract Moore machine. There are a

```
input set[1]
input start[8]
output count[8] := 0
output alarm[1] := 1

loop
    if set = 1
        count := start
    else if count > 0
        count := count − 1
    endif
    if count = 0
        alarm := 1
    else
        alarm := 0
    endif
    wait
endloop
```

Figure 2: An example program

number of abstractions built into the compiler, and the user may define new abstractions by supplying procedures to build the BDDs representing them. Abstract versions of the primitive relations are computed automatically.

Figure 2 is a small example program: a settable countdown timer. The timer has two inputs, *set* and *start*, which are one and eight bits wide respectively. There are also two outputs: *count*, which is eight bits wide and is initially zero; and *alarm*, which is one bit and initially one. At each time step, the operation of the counter is as follows. If *set* is one, then the counter is set to the value of *start*. Otherwise, if the counter is not zero, it is decremented. The *alarm* output is set to one when *count* is zero, and to zero if *count* is nonzero.

9 Example abstractions

In this section, we discuss some abstractions which have proved useful in practice. Each is illustrated with a small example. The temporal logic formulas in this section are written with some syntactic sugaring of the atomic propositions in order to make them easier to read.

9.1 Congruence modulo an integer

For verifying programs involving arithmetic operations, a useful abstraction is congruence modulo a specified integer m:

$$h(i) = i \bmod m.$$

This abstraction is motivated by the following properties of arithmetic modulo m.

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv i + j \pmod m$$
$$((i \bmod m) - (j \bmod m)) \bmod m \equiv i - j \pmod m$$
$$((i \bmod m)(j \bmod m)) \bmod m \equiv ij \pmod m$$

In other words, we can determine the value modulo m of an expression involving addition, subtraction and multiplication by working with the values modulo m of the subexpressions.

The abstraction may also be used to verify more complex relationships by applying the following result from elementary number theory.

Theorem 6 (Chinese remainder theorem) *If m_1, m_2, ..., m_n are positive integers which are pairwise relatively prime, $m = m_1 m_2 \ldots m_n$, and b, i_1, i_2, ..., i_n are integers, then there is a unique integer i such that for $1 \le j \le n$,*

$$b \le i \le b + m \qquad \text{and} \qquad i \equiv i_j \pmod{m_j}.$$

Suppose that we are able to verify that at a certain point in the execution of a program, the value of the nonnegative integer variable x is equal to i_j modulo m_j for each of the relatively prime integers $m_1, m_2, \ldots, m_n$. Further, suppose that the value of x is constrained to be less than $m_1 m_2 \ldots m_n$. Then using the above result, we can conclude that the value of x at that point in the program is uniquely determined.

We illustrate this abstraction using a 16 bit by 16 bit unsigned multiplier (see figure 3). The program has inputs *req*, *in1* and *in2*. The last two inputs provide the factors to operate on, and the first is a request signal which starts the multiplication. Some number of time units later, the output *ack* will be set to true. At that point, either *output* gives the 16 bit result of the multiplication, or *overflow* is one if the multiplication overflowed. The multiplier then waits for *req* to become zero before starting another cycle. The multiplication itself is done with a series of shift-and-add steps. At each step, the low order bit of the first factor is examined; if it is one, then the second factor is added to the accumulating result. The first factor is then shifted right and the result is shifted left in preparation for the next step.[1]

The specification we used for the multiplier was a series of formulas of the form[2]

$$\forall \mathbf{G}(\mathit{ready} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j)$$

[1] One feature of the language which the program uses is the ability to extend an operand to a specified number of bits. For example, $x{:}5$ extends x to be 5 bits wide by adding leading 0 bits. This facility is used to extend *output* and *factor2* when adding and shifting so that overflow can be detected. The statement $(\mathit{overflow}, \mathit{output}) := (\mathit{output}{:}17) + \mathit{factor2}$ sets *output* to the 16 bit sum of *output* and *factor2*, and *overflow* to the carry from this sum. Also, $x \ll 1$ is x shifted left by one bit. Right shifts are indicated using $\gg$. The `break` statement is used to exit the innermost loop.

[2] This specification admits the possibility that the multiplier always signals an overflow. We will verify that this is not the case using a different abstraction (see subsection 9.2).

```
input in1[16]
input in2[16]
input req
output factor1[16] := 0
output factor2[16] := 0
output output[16] := 0
output overflow := 0
output ack := 0

procedure waitfor(e)
    while ¬e
        wait
    endwhile
endproc

loop
  1: waitfor(req)
    factor1 := in1
    factor2 := in2
    output := 0
    overflow := 0
    wait
    loop
        if (factor1 = 0) ∨ (overflow = 1)
            break
        endif
        if factor1[0] = 1
            (overflow, output) :=
                (output: 17) + factor2
        endif
        factor1 := factor1 ≫ 1
        wait
        if (factor1 = 0) ∨ (overflow = 1)
            break
        endif
        (overflow, factor2) := (factor2: 17) ≪ 1
        wait
    endloop
    ack := 1
    wait
    waitfor(¬req)
    ack := 0
endloop
```

Figure 3: A 16 bit multiplier

$$\rightarrow \forall(\neg ack\ \mathbf{U}\ ack \wedge ok))$$

where

$$ok \equiv (overflow \vee (output \bmod m = ij \bmod m)).$$

Here, i and j range from 0 through $m-1$, and *ready* is an atomic proposition which is true when execution is at the program statement labeled 1. The input *in2* and the outputs *factor2* and *output* were all abstracted modulo m. The output *factor1* was not abstracted, since its entire bit pattern is used to control when *factor2* is added to *output*. We performed the verification for $m = 5$, 7, 9, 11 and 32. These numbers are relatively prime, and their product, 110,880, is sufficient to cover all 2^{16} possible values of *output*. The entire verification required slightly less than 30 minutes of CPU time on a Sun 4. We also note that because the BDDs needed to represent multiplication grow exponentially with the size of the multiplier, it would not have been feasible to verify the multiplier directly. Further, even checking the above formulas on the unabstracted multiplier proved to be impractical.

9.2 Representation by logarithm

When only the order of magnitude of a quantity is important, it is sometimes useful to represent the quantity by (a fixed precision approximation of) its logarithm. For example, suppose $i \geq 0$. Define

$$\lg i = \lceil \log_2(i+1) \rceil,$$

i.e., $\lg i$ is 0 if i is 0, and for $i > 0$, $\lg i$ is the smallest number of bits needed to write i in binary. We take $h(i) = \lg i$.

As an illustration of this abstraction, consider again the multiplier of figure 3. Recall that a program which always indicated an overflow would satisfy our previous specification. We note that if $\lg i + \lg j \leq 16$, then $\lg ij \leq 16$, and hence the multiplication of i and j should not overflow. Conversely, if $\lg i + \lg j \geq 18$, then $\lg ij \geq 17$, and the multiplication of i and j will overflow. When $\lg i + \lg j = 17$, we cannot say whether overflow should occur. These observations lead us to strengthen our specification to include the following two formulas.

$$\forall\mathbf{G}\big(ready \wedge req \wedge (\lg in1 + \lg in2 \leq 16) \rightarrow \forall(\neg ack\ \mathbf{U}\ ack \wedge \neg overflow)\big)$$

$$\forall\mathbf{G}\big(ready \wedge req \wedge (\lg in1 + \lg in2 \geq 18) \rightarrow \forall(\neg ack\ \mathbf{U}\ ack \wedge overflow)\big)$$

We represented all the 16 bit variables in the program by their logarithms. Compiling the program with this abstraction and checking the above properties required less than a minute of CPU time.

9.3 Single bit and product abstractions

For programs involving bitwise logical operations, the following abstraction is often useful:

$$h(i) = \text{the } j\text{th bit of } i,$$

where j is some fixed number.

If h_1 and h_2 are abstraction mappings, then

$$h(i) = (h_1(i), h_2(i))$$

also defines abstraction mapping. Using this abstraction, it may be possible to verify properties that it is not possible to verify with either h_1 or h_2 alone.

```
input in[16]
output parity[1] := 0
output b[16] := 0
output done[1] := 0

b := in
wait
while b ≠ 0
    parity := parity ⊕ b[0]
    b := b ≫ 1
    wait
endwhile
done := 1
```

Figure 4: A parity computation program

As an example of using these types of abstractions, consider the program shown in figure 4. This program reads an initial 16 bit input and computes the parity of it. The output *done* is set to one when the computation is complete; at that point, *parity* has the result. Let $\sharp i$ be true if the parity of i is odd. One desired property of the program is the following.

1. The value assigned to b has the same parity as that of *in*; and
2. $parity \oplus \sharp b$ is invariant from that point onwards.

We can express the above with the following formula.

$$\neg\sharp in \wedge \forall\mathbf{X}\big(\neg\sharp b \wedge \forall\mathbf{G}\,\neg(parity \oplus \sharp b)\big)$$
$$\vee\, \sharp in \wedge \forall\mathbf{X}\big(\sharp b \wedge \forall\mathbf{G}(parity \oplus \sharp b)\big)$$

To verify this property, we used a combined abstraction for *in* and b. Namely, we grouped the possible values for these variables both by the value of their low order bit and by their parity. The verification required only a few seconds.

9.4 Symbolic abstractions

The use of a BDD-based compiler together with model checker makes it possible to use abstractions which depend on symbolic values. This idea can greatly increase the power of a particular type of abstraction. For example, consider a simple partitioning:

$$h(i) = \begin{cases} 0, & \text{if } i < a; \\ 1, & \text{if } i \geq a. \end{cases}$$

where a is some fixed value. We might try to use such an abstraction when the program we are trying to verify involves comparisons. If two numbers are not equivalent according to this abstraction, we can find the truth value of a comparison between them. In most cases however, using only this single abstraction would not imply much about the unabstracted program's behavior. Much more information may be obtained by letting a be a symbolic constant. Using such an abstraction allows us to verify that a formula is true for all possible abstract programs obtainable by varying a. Thus, we can effectively verify an entire class of properties for the unabstracted program.[3]

As an example of using this abstraction, consider the program of figure 5. This program represents a cell in a linear sorting array. There is one cell for each integer to be sorted, and the cells are numbered consecutively from right to left. In the array, each cell's *left* and *leftsorted* inputs are connected to its left neighbor's y and *sorted* outputs, and each cell's *right* input is connected to its right neighbor's x output. The values to be sorted are the values of the x outputs. The sort proceeds in cycles. During each cycle, exactly half the cells (either all the odd numbered cells or all the even numbered cells) will have their *comparing* output equal to one. These cells compare their own x output with that of their right neighbor. The smaller of these values is placed in y. In addition, if the values were swapped, the cell's *sorted* output is set to zero. During the next clock period, the right neighbor's x and *sorted* values are copied from the first cell's y and *sorted* outputs. When the rightmost cell's *sorted* output becomes one, the sort is complete. In this example, we consider an array for sorting eight numbers.[4]

The properties which we verified are:

1. for every a, eventually the values of the x outputs are such that all numbers which are less than a come before all numbers which are greater than or equal to a, and this condition holds invariantly from that point on; and
2. for every a, the number of the x outputs which are less than a is invariant except when elements are being swapped.

The first property implies that the array is eventually sorted. The second one implies that the final values of the x outputs form a permutation of the initial values.

We performed the verification by abstracting all the 16 bit variables in the program as described above. The temporal formulas corresponding to the two properties are

$$\forall\mathbf{F}\,\forall\mathbf{G}\big((x[1] < a \vee x[2] \geq a) \wedge \cdots \wedge (x[7] < a \vee x[8] \geq a)\big)$$

[3] In our compiler, non-symbolic abstractions are specified by giving a relation $H(d, d')$ which represents $h(d) = d'$. For a symbolic abstraction, this relation is extended with additional parameters which are the symbolic constants it depends on. The BDD representation of the final Moore machine will depend on these symbolic constants. The model checker simply treats the symbolic constants as additional state variables.

[4] In this program, x and y may have any initial values. The *comparing* output is set to zero or one depending on the cell's position in the array. The left and right ends of the sorting array are dummy cells for which x is $2^{16} - 1$ and 0 respectively. The left cell's *sorted* output is also fixed at 1.

```
input left[16]
input leftsorted[1]
output sorted[1] := 0
output comparing[1] := 0 or 1
output swap[1] := 0
output x[16]
output y[16]
input right[16]

loop
    if comparing = 1
        swap := (x < right)
        wait
        if swap = 1
            y := x
            x := right
            sorted := 0
        else
            y := right
        endif
        wait
    else
        wait
        wait
        x := left
        sorted := leftsorted
    endif
    comparing := ¬comparing
  1: wait
endloop
```

Figure 5: A sorting cell program

and

$$\left(\sum_{i=1}^{8}(x[i] < a) = n\right) \rightarrow \forall\mathbf{G}\left(\left(\sum_{i=1}^{8}(x[i] < a) = n\right) \vee \neg stable\right).$$

Here, the summation denotes the number of formulas $x[i] < a$ which are true, and *stable* is an atomic proposition which is true when every cell is executing the statement labeled 1.[5] Verifying these properties required just under five minutes of CPU time. In addition, checking these properties on the unabstracted program was not feasible due to space limitations.

We also used symbolic abstractions to verify a simple pipeline circuit. This circuit is shown in figure 6 and is described in detail elsewhere [3, 4]. It performs three-address arithmetic and logical operations on operands stored in a register file.

We used two independent abstractions to perform the verification. First, the register addresses were abstracted so that each address was either one of three symbolic constants (*ra*, *rb* or *rc*) or some other value.

[5] We also verified the property $\forall\mathbf{G}\forall\mathbf{F}\, stable$ to check that the cells maintain lockstep.

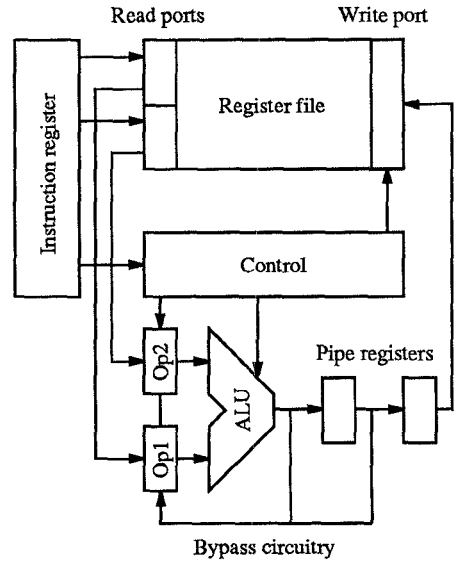


Figure 6: Pipeline circuit block diagram

This abstraction made it possible to collapse the entire register file down to only three registers, one for each constant. The second abstraction involved the individual registers in the system. In order to verify an operation, say addition, we create symbolic constants *ca* and *cb* and allow each register to be either *ca*, *cb*, *ca*+*cb* or some other value. As part of the specification, we verified that the circuit's addition operation works correctly. This property is expressed by the temporal formula

$$\forall\mathbf{G}\big((src1 = ra) \wedge (src2 = rb) \wedge (dest = rc) \wedge \neg stall \rightarrow \forall\mathbf{X}\forall\mathbf{X}((regra = ca) \wedge (regrb = cb) \rightarrow \forall\mathbf{X}(regrc = ca + cb))\big).$$

This formula states that if the source address registers are *ra* and *rb*, the destination address register is *rc*, and the pipeline is not stalled, then the values in registers *ra* and *rb* two cycles from now will sum to the value in register *rc* three cycles from now. The reason for using the values of registers *ra* and *rb* two cycles in the future is to account for the latency in the pipeline.

The largest pipeline example we tried had 64 registers in the register file and each register was 64 bits wide. This circuit has more than 4,000 state bits and nearly 10^{1300} reachable states. The verification required slightly less than six and one half hours of CPU time. In addition the verification times scale linearly in both the number of registers and the width of the registers. For comparison, the largest circuit verified by Burch *et al.* [3] had 8 registers, each 32 bits, and the verification required about four and one half hours of CPU time on a Sun 4. In addition the verification times there were growing quadratically in the register width and cubically in the number of registers.

10 Conclusion

We have described a simple but powerful method for using abstraction to simplify the problem of model checking. There are two parts to this method. First, we have shown how to extract abstract finite state machines directly from finite or infinite state programs. The construction guarantees that the actual state machine for the program is a refinement of the extracted state machine. Second, we have examined when satisfaction of a formula by an abstract machine implies satisfaction by the actual machine. For formulas given in the logic $\forall$CTL*, this is always the case. We have also implemented a symbolic verification system based on these ideas and used it to verify a number of nontrivial examples. In the process of doing these examples, we have found a number of useful abstractions. Our work on generating abstract systems could be used with other verification methodologies, such as testing language containment, as well.

There are a number of possible directions for future work. One problem with using our current approach with logics like CTL*, which can express the existence of a path, is in ensuring the strict exactness conditions. By using a more complex finite state model such as AND/OR graphs, it should be possible to extend the techniques and obtain a conservative model checking algorithm for such logics. We also wish to explore thoroughly the problem of generating abstractions for infinite state systems. The important step in doing this is determining abstract versions of the primitive relations. Some of the techniques and results from automated theorem proving, term rewriting, and algebraic specification of abstract data types should prove useful for this problem. Similar techniques would be useful for studying the flow of data in a system. Data items might be represented as terms in the Herbrand universe and functional transformations on the data would correspond to building new terms from the input terms. Given an equivalence relation of finite index on terms, we would derive abstract primitive relations for the operations and use these to produce an abstract version of the system.

References

[1] 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.

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

[3] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. 28th ACM/IEEE Design Automation Conf.* IEEE Comp. Soc. Press, June 1991.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th ACM/IEEE Design Automation Conf.* IEEE Comp. Soc. Press, June 1990.

[5] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proc. 10th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1983.

[7] 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.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. 4th Ann. Symp. on Logic in Comput. Sci.* IEEE Comp. Soc. Press, June 1989.

[9] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. 1990 IEEE Inter. Conf. on Comput.-Aided Design.* IEEE Comp. Soc. Press, Nov. 1990.

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

[11] S. Graf and B. Steffen. Compositional minimization of finite state processes. In R. P. Kurshan and E. M. Clarke, editors, *Proc. 1990 Workshop on Comput.-Aided Verification*, June 1990.

[12] O. Grumberg and D. E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *Proc. CONCUR '91: 2nd Inter. Conf. on Concurrency Theory*, volume 527 of *LNCS*. Springer-Verlag, Aug. 1991.

[13] Z. Har'El and R. P. Kurshan. The COSPAN user's guide. Technical Report 11211-871009-21TM, AT&T Bell Labs, 1987.

[14] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *LNCS*. Springer-Verlag, May 1989.

[15] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1985.

[16] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Fifth Inter. Symp. in Programming*, 1981.

[17] A. P. Sistla and E. Clarke. Complexity of propositional temporal logics. *J. ACM*, 32(3):733–749, July 1986.

[18] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1986.