Model Checking and Abstraction

EDMUND M. CLARKE
Carnegie Mellon University
ORNA GRUMBERG
The Technion
and
DAVID E. LONG
AT&T Bell Laboratories

We describe a method for using abstraction to reduce the complexity of temporal logic model checking. Using techniques similar to those involved in abstract interpretation, we construct 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 program representing a pipelined ALU circuit with over 10^{1300} states.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*

General terms: Verification
Additional Key Words and Phrases: Temporal logic, model checking, abstract interpretation, binary decision diagrams (BDDs)

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 [8; 13; 29; 33; 35] is a method for automatically deciding if a finite state program satisfies its specification. A model

Author's addresses: E. M. Clarke, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; O. Grumberg, Department of Computer Science, The Technion, Haifa, Israel 32000; D. E. Long, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.
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 in part by the U.S.-Israeli Binational Science Foundation. The views and conclusions contained in this document are those of the authors should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the U.S. Government.

checking algorithm for the propositional branching time temporal logic CTL was presented at the 1983 POPL conference [9]. The algorithm was linear both in 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 eleven 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 [4]. Use of binary decision diagrams (BDDs) [5] 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 of the algorithm [7]. Refinements of the BDD-based techniques [6] 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 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 $\widehat{D}_i$ 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 *minimal abstraction* 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 very large, this may not be feasible. Even if it is possible to represent the system using BDD-based methods, the computational complexity of building the minimal abstraction may still be very high. To circumvent these problems, we show how to derive an *approximation* to the minimal 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.

This symbolic execution is exactly the same idea as is used in *abstract interpretation* as pioneered by the Cousots [15, 16]. In the Cousots' work, the spaces of concrete and abstract data values are complete lattices (or more generally, complete partial orders). The relation between levels is given by a Galois connection (α, γ). α maps concrete values to abstract values, and γ maps back. The mapping h above is the analog of α and its inverse would correspond to γ. In abstract interpretation, given (α, γ) and a programming language semantics, we derive an abstract semantics for the language. Our symbolic execution corresponds to evaluating a program under this abstract semantics. The effect of the evaluation is to produce directly an abstract representation of the program's behavior. The differences between our work and most of the work on abstract interpretation are summarized below. These differences arise mainly from the differing applications of the work. Most abstract interpretations are designed to collect information about the static semantics of a program (typically for use by an optimizing compiler). The static semantics gives information about all of the possible program states at a given program point. Hence it is useful for answering questions about live variables, available expressions, etc. Further, since compilers must deal with very large programs, the emphasis is often on trading accuracy for speed in the analysis. In contrast, we are interested in the dynamic behavior of the program (the transitions between states), and proving the correctness of a system generally requires a precise analysis. Because of these strict requirements, we cannot handle very large programs.

(1) In our work, producing an abstract model of the system is only the first step in the verification process. Afterwards, we use state space searches to check temporal properties.
(2) In abstract interpretation, the abstractions are usually defined with a particular type of analysis in mind and then fixed. Hence, constructing the abstract version of the language semantics can be done once, and with manual assistance. In verification, the user often needs to define new abstractions "on the fly." This need arises because of the delicate balance between keeping enough information to have the verification go through, and throwing out enough to keep the time and space requirements reasonable. Having to produce a new abstract semantics by hand for each new abstraction would be extremely tedious. As a result, our tools must do this automatically. However, to ensure decidability, we have to restrict ourselves to finite data domains.
(3) Because of the need to be precise, we always view expressions as evaluating (at the abstract level) to some set of possible abstract values. (This set could be mapped back to a set of possible concrete values.) In abstract interpretation, this would correspond to working over a powerdomain [23]. However, in the abstract model that we construct, states are simply assignments of single abstract values to the program variables. This corresponds more to a flat domain. Because we always use this same type of interpretation, we can eliminate many of the technical details that would otherwise be necessary to translate back and forth between the different types of domains.

Recently, Bensalem *et al.* [3] have considered abstractions as Galois connections between sets of states of two processes. They then consider the relationship between abstract-level and concrete-level satisfaction of logical properties expressed in a fixpoint calculus. Their notation is close to that used in the abstract interpretation literature, while ours is most similar to that in earlier work on using abstraction for finite-state verification (e.g., [27]).

The specification language that we use is a propositional temporal logic called CTL* [10]. 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 our approach is *conservative* if we restrict to a subset of the logic called ∀CTL* [22] in which only the ∀ path quantifier is allowed. That is, 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 note 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 in 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 [21] 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 [12] describe a related attempt. Grumberg and Long [22] and Shurek and Grumberg [34] propose frameworks for compositional verification based on ∀CTL*. Dill [18] has developed a trace theory for compositional design of asynchronous circuits. 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 [37] 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 [7; 14] 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 6, 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 [27] is most closely related to ours. This approach has been automated in the COSPAN system [24]. 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 homomorphisms induced by boolean algebra homomorphisms in Kurshan's work. 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.
(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 and the notion of abstraction that we use. Section 4 discusses constructing an approximate abstract transition system directly from a program. It also discusses the conditions required for exactness. Section 5 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 note that if the approximation is exact, then all CTL* formulas are preserved. Section 6 describes a programming language that can be used for specifying finite-state systems, and describes the verification of several systems via a variety of abstractions. The paper concludes with a discussion of some directions for future research.

2. BINARY DECISION DIAGRAMS

Binary decision diagrams (BDDs) are a canonical form representation for boolean formulas described by Bryant [5]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of CAD applications, including symbolic simulation [2], verification of combinational logic [20], and verification of sequential circuits [7; 14; 36]. A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Consider, for example, the BDD of figure 1. It represents the formula $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of boolean values to the variables a, b, c and d, one can decide whether the assignment makes the formula true by traversing the graph beginning at the root and branching at each node based on the value assigned to the variable that labels the node. For example, the

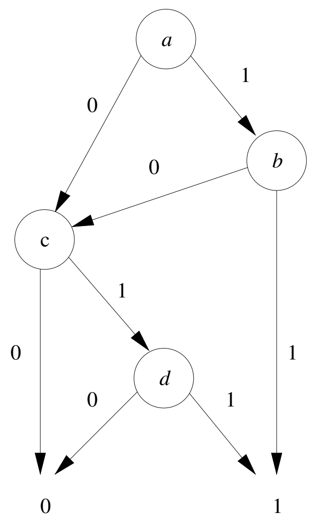

Fig. 1. A BDD representing $(a \wedge b) \vee (c \wedge d)$

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. The size of the BDD depends critically on the variable ordering. Bryant 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.

Another way to view BDDs is as deterministic finite automata (DFAs) [11]. The initial state of the automata is the root of the BDD, and the only accepting state is the terminal 1. From this viewpoint, the BDD operations correspond to standard constructions such as language intersection and union for DFAs. The canonical form property of BDDs corresponds to the uniqueness of the minimal DFA accepting a given language.

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 [7; 14; 36].

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 (non-empty) 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 (non-empty) sets $\widehat{D}_i$ 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 $h_i : D_i \rightarrow \widehat{D}_i$ for each i. These mappings induce a surjection $h : D \rightarrow \widehat{D}$ 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 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. We also note that

$$(d_1, \ldots, d_n) \sim (e_1, \ldots, e_n) \qquad \text{if and only if} \qquad d_1 \sim_1 e_1 \wedge \cdots \wedge d_n \sim_n e_n.$$

We will sometimes specify abstractions by mappings and sometimes specify them by equivalence relations.

Let M be a transition system over D and let h be a surjection from D to $\widehat{D}$. We now define what it means for a transition system over the abstract set of states $\widehat{D}$ to be an abstract version of M. The intuition is that a state $\widehat{s}$ of the abstract system will represent all those states s of M for which $h(s) = \widehat{s}$. The abstract state $\widehat{s}$ must be able to simulate each such s, so if s can transition to s', then we will require that $\widehat{s}$ be able to transition to $\widehat{s'} = h(s')$. Similarly, if M could start in state s, we require that the abstract system be able to start in $\widehat{s}$. Formally, we have the following definition.

Definition 2. Let $\widehat{M}$ be a transition system over $\widehat{D}$. We say that $\widehat{M}$ *approximates* M (denoted $M \sqsubseteq_h \widehat{M}$) when:

(1) $\exists d\,(h(d) = \widehat{d} \wedge I(d))$ implies $\widehat{I}(\widehat{d})$.
(2) $\exists d_1 \exists d_2\,(h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$ implies $\widehat{R}(\widehat{d_1}, \widehat{d_2})$.

There is a natural abstract transition system having only those initial states and transitions required by the above definition. We call this "minimal" transition system $\widehat{M}_{\min}$.

Definition 3. $\widehat{M}_{\min}$ is the transition system over $\widehat{D}$ given by:

(1) $\widehat{I}_{\min}(\widehat{d})$ if and only if $\exists d\,(h(d) = \widehat{d} \wedge I(d))$.

(2) $\widehat{R}_{\min}(\widehat{d}_1, \widehat{d}_2)$ if and only if $\exists d_1 \exists d_2\,(h(d_1) = \widehat{d}_1 \wedge h(d_2) = \widehat{d}_2 \wedge R(d_1, d_2))$.

Obviously $M \sqsubseteq_h \widehat{M}_{\min}$. Further, for any other transition system $\widehat{M}$ over $\widehat{D}$, we see that $M \sqsubseteq_h \widehat{M}$ if and only if $\widehat{I} \supseteq \widehat{I}_{\min}$ and $\widehat{R} \supseteq \widehat{R}_{\min}$. Thus, $\widehat{M}_{\min}$ is the most accurate approximation to M that is consistent with h.

As we will show in section 5, an abstract transition system such as $\widehat{M}_{\min}$ may be used to deduce properties of M.[1] 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 $\widehat{M}_{\min}$ 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 produce BDDs representing I and R, computing BDDs representing $\widehat{I}_{\min}$ and $\widehat{R}_{\min}$ requires a number of relational products (essentially, one for each h_i when computing the BDD for $\widehat{I}_{\min}$ and two for each h_i when computing the BDD for $\widehat{R}_{\min}$). In practice, we have found that evaluating these relational products is often impossible. In the next section, we discuss a method for circumventing these problems. This method is based on the fact that we usually have an *implicit* representation of M as a program in a finite-state language. We will show how to compute an approximation to M directly from the program text. Hence, it is never necessary to construct BDDs representing I and R. In addition, we demonstrate empirically that the approximation is generally accurate enough to allow us to verify interesting properties of the program. Note that in the abstract interpretation literature, it is generally the approximation that is highlighted, while $\widehat{M}_{\min}$ is often implicit. However, from a conceptual point of view, we would like to produce an abstraction that is as close as possible to $\widehat{M}_{\min}$.

4. PRODUCING ABSTRACT MODELS

In this section, we consider the problem of deriving an approximate abstract model of M directly from a finite-state program describing M. The actual process will be described in subsection 4.2. However, we would like this discussion to be relatively independent of the particular finite-state language used. To accomplish this, we are going to argue that a program in a finite-state language can be transformed into *relational expressions* $\mathcal{I}$ and $\mathcal{R}$ that can be evaluated to obtain the initial states I and the transition relation R of the transition system M represented by the program. These relational expressions are simply formulas in first-order predicate logic that will built up from a set of *primitive relations* for the basic operators and constants in the language. Then in subsection 4.2, we will show how to manipulate $\mathcal{I}$ and $\mathcal{R}$ to obtain the approximation to M. There will typically be types associated with the variables and relation arguments in the relational expressions that we write, but for notational simplicity, we will leave these implicit.

[1] The reader may be concerned about eliminating deadlocks by adding new initial states and transitions. This is discussed in section 5.

4.1 Semantics of finite-state programs

In this subsection, we consider how $\mathcal{I}$ and $\mathcal{R}$ can be derived. Since this is not the main concern of the paper, we will just consider an example program (figure 2). This program computes the parity p of the variable b by repeatedly computing the exclusive-or of p and the low-order (rightmost) bit of b ($\mathrm{lsb}(b)$) and then shifting b to the right by one bit ($b \gg 1$). (The parity of a number is 0 if the number of one bits in its binary representation is even, and 1 if this number is odd.) Since we are interested in verifying the temporal behavior of programs, we must know the points where the state of the variables can be observed. We will call these points *control points*, and in the example, the control points are those lines labeled with 0, 1, and 2. During the computation of this program, we will observe a transition from control point 0 to control point 1 (during which p is set to 0), some transitions from 1 back to 1 (going around the while loop), a transition from 1 to 2 (when $b = 0$), and finally an infinite sequence of transitions from 2 to 2 when the program is in a terminal state. (We add loops at terminal states since our specification logic only describes infinite behaviors.) Contrast this with the input-output style semantics of the program, where we would just be interested in the relationship between the variables at points 0 and 2. Looking at the state transitions between control points is also the basis of program verification techniques such as the *inductive assertion method* [19].

0: $p := 0$
1: **while** $b \neq 0$
        $p := p \oplus \mathrm{lsb}(b)$
        $b := b \gg 1$
   endwhile
2: **end**

Fig. 2. A simple example program

The transition relation specified by this program is obtained by looking at the sequences of statements between consecutive control points. First, consider the transition between control points 0 and 1. During this transition, p should be set to 0. To distinguish the values of the variables at the start of the transition (at control point 0) from the values at the end of the transition (at 1), we will decorate the latter with primes. Thus, p will denote the value of the variable p at point 0, and p' will denote the value of the variable p at point 1. We will use a variable PC ("Program Counter") to denote the control point. Then the transition from point 0 to point 1 can be expressed by:

$$\mathrm{PC} = 0 \wedge p' = 0 \wedge b' = b \wedge \mathrm{PC}' = 1.$$

This says that PC starts at 0 and ends at 1, the value of p at the end point is 0, and the value of b does not change during the transition.

The transition from point 1 to point 2 does not involve any changes in the variables, but it does require a test to see that $b = 0$. Thus, we get the relation:

$$\mathrm{PC} = 1 \wedge b = 0 \wedge p' = p \wedge b' = b \wedge \mathrm{PC}' = 2.$$

The $b = 0$ acts as a guard to eliminate the transition when the condition does not hold. An expression for the transition relation of the whole program can be derived by simply taking the disjunction of the expressions for the point-to-point transitions. For this program, we get the following expression (the first two lines are just the point to point relations derived above):

$$\begin{gathered}(\mathrm{PC} = 0 \wedge p' = 0 \wedge b' = b \wedge \mathrm{PC}' = 1) \vee \\ (\mathrm{PC} = 1 \wedge b = 0 \wedge p' = p \wedge b' = b \wedge \mathrm{PC}' = 2) \vee \\ (\mathrm{PC} = 1 \wedge b \neq 0 \wedge p' = p \oplus \mathrm{lsb}(b) \wedge b' = b \gg 1 \wedge \mathrm{PC}' = 1) \vee \\ (\mathrm{PC} = 2 \wedge p' = p \wedge b' = b \wedge \mathrm{PC}' = 2).\end{gathered}$$

Note that in this program, the loop is broken by a control point (point 1). For simplicity, we will assume that this is always the case. However, since we will only be working over finite domains, it is not strictly necessary. That is, we could allow unbroken loops between control points and then check that such loops always terminated.

The above expression is written assuming that we have operators in the logic for all of the operators in the language, that we can use language constants as constants in the logic, etc. To eliminate these, we could instead rewrite the above expression in terms of *primitive relations* for the operators and constants. Consider, for example, the clause $p' = p \oplus \mathrm{lsb}(b)$. This involves two operations: selecting the low-order bit of b, and then computing the exclusive-or of the result with p. We now assume that we have primitive relations P_{lsb} and $P_{\oplus}$ for these operators. The former is a two-argument relation, and the latter is a three-argument relation: the last argument in each case will be the result produced by the operator. The clause $p' = p \oplus \mathrm{lsb}(b)$ can now be expressed as follows:

$$\exists t\, (P_{lsb}(b, t) \wedge P_{\oplus}(p, t, p')).$$

(Note that we needed to introduce a "temporary" variable t to hold the intermediate result.) In a similar way, we could rewrite the rest of the transition relation expression to obtain a relational expression built entirely from primitive relations. This would be the relational expression $\mathcal{R}$. A relational expression $\mathcal{I}$ describing the initial conditions on p, b, and PC could be derived in a similar way.

In general, the derivation of $\mathcal{I}$ and $\mathcal{R}$ is based on a *relational semantics* for the finite-state language: essentially, we write down the meaning of the program under the semantics. A relational semantics is usually very natural for languages intended to specify transition systems since their purpose is to describe the transition relation of the system. We will not give the relational semantics for any particular language in this paper; our goal above is just to motivate the claim that we can take a finite-state program and produce relational expressions representing the initial states and transitions of the transition system described by the program.

4.2 Computing approximations

In the previous subsection, we argued that the initial states and transition relation of a transition system M could be represented by formulas $\mathcal{I}$ and $\mathcal{R}$. Similar formulas $\widehat{\mathcal{I}}_{\mathrm{min}}$ and $\widehat{\mathcal{R}}_{\mathrm{min}}$ can be obtained representing $\widehat{M}_{\mathrm{min}}$. Since actually evaluating $\widehat{\mathcal{I}}_{\mathrm{min}}$ and $\widehat{\mathcal{R}}_{\mathrm{min}}$ can be computationally complex, we now show how to obtain formulas $\widehat{\mathcal{I}}_{\mathrm{app}}$ and $\widehat{\mathcal{R}}_{\mathrm{app}}$ describing an approximation $\widehat{M}_{\mathrm{app}}$ to M. Throughout this subsection and the next, we assume that ϕ, ϕ_1 and ϕ_2 are relational expressions built up from the primitive relations representing the operations in the program. For simplicity, we assume that all of the variables x_1, x_2, …, range over the same domain D. We also use a set $\widehat{x_1}$, $\widehat{x_2}$, …, of variables ranging over the abstract domain $\widehat{D}$, with $\widehat{x_i}$ representing the abstract value of x_i. We will also assume that there is only one abstraction function h mapping elements of D to elements of $\widehat{D}$. (Note that we are abusing notation a bit, since D, $\widehat{D}$ and h are also used to denote the (product) concrete and abstract state spaces and the mapping between these state spaces.)

Recall that building $\widehat{M}_{\text{min}}$ 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 $[\cdot]$. If ϕ depends on the free variables x_1, …, x_m, then we define

$$[\phi](\widehat{x_1},\ldots,\widehat{x_m}) = \exists x_1 \ldots \exists x_m \,(h(x_1)=\widehat{x_1} \wedge \cdots \wedge h(x_m)=\widehat{x_m} \wedge \phi(x_1,\ldots,x_m)).$$

Note that the free variables of $[\phi]$ are the abstract versions of x_1, …, x_m. Based on the definition of $\widehat{M}_{\text{min}}$, we observe that if $\mathcal{I}$ and $\mathcal{R}$ are the formulas representing I and R, then $\widehat{\mathcal{I}}_{\text{min}} = [\mathcal{I}]$ and $\widehat{\mathcal{R}}_{\text{min}} = [\mathcal{R}]$ are formulas representing $\widehat{I}_{\text{min}}$ and $\widehat{R}_{\text{min}}$.

Ideally, we would like to evaluate $[\mathcal{I}]$ and $[\mathcal{R}]$ directly. However, applying $[\cdot]$ to complex formulas can be computationally expensive. Thus, we will now define a transformation $\mathcal{T}$ on formulas ϕ. The idea of $\mathcal{T}$ is 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, then $\mathcal{T}(P(x_1,\ldots,x_m)) = [P](\widehat{x_1},\ldots,\widehat{x_m})$ and $\mathcal{T}(\neg P(x_1,\ldots,x_m)) = [\neg P](\widehat{x_1},\ldots,\widehat{x_m})$.
(2) $\mathcal{T}(\phi_1 \wedge \phi_2) = \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)$.
(3) $\mathcal{T}(\phi_1 \vee \phi_2) = \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2)$.
(4) $\mathcal{T}(\forall x\, \phi) = \forall \widehat{x}\, \mathcal{T}(\phi)$.
(5) $\mathcal{T}(\exists x\, \phi) = \exists \widehat{x}\, \mathcal{T}(\phi)$.

In other words, $\mathcal{T}$ applies the operation $[\cdot]$ only at the innermost level. Since these inner formulas are relatively simple, they can be evaluated easily. We can now produce the transition system $\widehat{M}_{\text{app}}$ by evaluating the formulas $\mathcal{T}(\mathcal{I})$ and $\mathcal{T}(\mathcal{R})$. However, to be able to use $\widehat{M}_{\text{app}}$ for verification purposes, we must ensure that we have not omitted any behaviors of the abstract system. That is, we must check that every transition of $\widehat{M}_{\text{min}}$ is also a transition of $\widehat{M}_{\text{app}}$, and that every initial state of $\widehat{M}_{\text{min}}$ is also an initial state of $\widehat{M}_{\text{app}}$. To do this, we examine the relationship between $[\phi]$ and $\mathcal{T}(\phi)$.

THEOREM 1. *$[\phi]$ implies $\mathcal{T}(\phi)$. In particular, $[\mathcal{I}]$ implies $\mathcal{T}(\mathcal{I})$ and $[\mathcal{R}]$ implies $\mathcal{T}(\mathcal{R})$. (The converse does not hold in general: in cases 2 and 4 above, $\mathcal{T}$ pushes existential quantifications over conjunctions, leading to inequivalent formulas.)*

PROOF. We apply induction on the structure of the formula ϕ.

(1) If $\phi = P(x_1,\ldots,x_m)$ or $\phi = \neg P(x_1,\ldots,x_m)$ where P is a primitive relation then $[\phi] = \mathcal{T}(\phi)$ and the lemma holds.

(2) Let $\phi(x_1, \ldots, x_m) = \phi_1 \wedge \phi_2$. ($\phi_1$ and ϕ_2 should be assumed to have the same parameter lists as ϕ, but for conciseness, we omit them.) Then, $[\phi_1 \wedge \phi_2]$ is identical to the formula

$$\exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_1 \wedge \phi_2).$$

This formula implies

$$\exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_1) \wedge \exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_2),$$

which is exactly $[\phi_1] \wedge [\phi_2]$. Now $\mathcal{T}(\phi_1 \wedge \phi_2) = \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)$, and by the induction hypothesis we have $[\phi_1]$ implies $\mathcal{T}(\phi_1)$ and $[\phi_2]$ implies $\mathcal{T}(\phi_2)$. Hence $[\phi_1] \wedge [\phi_2]$ implies $\mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)$, and so $[\phi_1 \wedge \phi_2]$ implies $\mathcal{T}(\phi_1 \wedge \phi_2)$.

(3) The case where $\phi = \phi_1 \vee \phi_2$ is similar to the previous case. (Note though that pushing the abstraction over a disjunction does not cause us to lose any information.)

(4) Let $\phi(x_1, \ldots, x_m) = \forall x \, \phi_1$. Then $[\forall x \, \phi_1]$ is

$$\exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \forall x \, \phi_1(x, x_1, \ldots, x_m)).$$

We can assume without loss of generality that the bound variable x is different from the x_i and $\widehat{x}_i$, so the above formula is equivalent to

$$\exists x_1 \ldots \exists x_m \forall x \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_1(x, x_1, \ldots, x_m)).$$

This implies

$$\forall x \exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_1(x, x_1, \ldots, x_m)).$$

Since h is a surjection, for every abstract element in $\widehat{D}$, there is some element of D that maps onto it. Hence the above formula implies

$$\forall \widehat{x} \exists x [\exists x_1 \ldots \exists x_m \, (h(x) = \widehat{x} \wedge \bigwedge_i h(x_i) = \widehat{x}_i \wedge \phi_1(x, x_1, \ldots, x_m))].$$

This is exactly $\forall \widehat{x} \, [\phi_1]$. Now by the induction hypothesis, $[\phi_1]$ implies $\mathcal{T}(\phi_1)$, and so $\forall \widehat{x} \, [\phi_1]$ implies $\forall \widehat{x} \, \mathcal{T}(\phi_1)$. This latter formula is equal to $\mathcal{T}(\forall x \, \phi_1)$.

(5) The case where $\phi = \exists x \, \phi_1$ is similar to the previous case. (Although as with disjunction, we do not lose information by pushing an abstraction over an existential quantification.) $\square$

The above idea of "pushing the abstractions inwards" is the same idea that is used in abstract interpretation [15; 16; 31; 32]. In abstract interpretation, when defining the abstract semantics induced by an abstraction, the meaning of part of the program (say an expression) in the programming language is given in terms of a composition of abstract versions of the operators in the language. Our abstract primitive relations correspond exactly to these abstract operators. Note that in general though, we will be producing these abstract primitive relations automatically based on the user-supplied abstraction mappings.

To be able to use $\widehat{M}_{\text{app}}$ for verification purposes, we want to know that the relation $\sqsubseteq_h$ holds between M and $\widehat{M}_{\text{app}}$. Then we will show in section 5 that every formula that is true for $\widehat{M}_{\text{app}}$ is also true for M.

THEOREM 2. *Let $\widehat{M}_{\text{app}}$ be the transition system obtained by evaluating $\mathcal{T}(\mathcal{I})$ and $\mathcal{T}(\mathcal{R})$. Then $M \sqsubseteq_h \widehat{M}_{\text{app}}$.*

PROOF. We know $M \sqsubseteq_h \widehat{M}_{\text{min}}$. By the previous theorem, $\widehat{I}_{\text{min}} \subseteq \widehat{I}_{\text{app}}$ and $\widehat{R}_{\text{min}} \subseteq \widehat{R}_{\text{app}}$. We also have $\widehat{S}_{\text{min}} = \widehat{S}_{\text{app}}$. By the definition of $\sqsubseteq_h$, these facts trivially imply $M \sqsubseteq_h \widehat{M}_{\text{app}}$. $\square$

4.3 Exact approximations

Above, we demonstrated that $M \sqsubseteq_h \widehat{M}_{\text{min}}$ and $M \sqsubseteq_h \widehat{M}_{\text{app}}$. These results will be used to show that our verification methodology is conservative. In this subsection, we make a note of some additional properties that suffice to make the method exact. By "exact", we mean that a property will be true at the concrete level if *and only if* it is true at the abstract level. Thus, the concrete and abstract models exhibit identical behavior in an appropriate sense. In our experience, requiring an exact approximation to M generally allows very little simplification, and hence exact approximations are not very useful for reducing the complexity of verification. For this reason, we will omit most of the details and proofs in this subsection. 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

$$\forall d_1 \ldots \forall d_m \forall e_1 \ldots \forall e_m \, (\bigwedge_j d_j \sim_{i_j} e_j \rightarrow (P(d_1, \ldots, d_m) \Leftrightarrow P(e_1, \ldots, e_m))).$$

If the $\sim_i$ are congruences with respect to the primitive relations, then $\widehat{M}_{\text{app}}$ is an exact approximation of M. This can be shown in two steps: first, $\widehat{M}_{\text{min}} = \widehat{M}_{\text{app}}$, and second, $\widehat{M}_{\text{min}}$ is an exact approximation of M. As in the previous subsection, we will simplify notation by assuming that all variables range over the same domain D, that there is one abstract domain $\widehat{D}$, and that there is one abstraction mapping h with corresponding equivalence relation $\sim$.

LEMMA 1. *If $\sim$ is a congruence with respect to the primitive relations then $[\phi] \Leftrightarrow \mathcal{T}(\phi)$.*

THEOREM 3. *If $\sim$ is a congruence with respect to the primitive relations, then $\widehat{M}_{\text{min}} = \widehat{M}_{\text{app}}$.*

Now we make precise what it means for one transition system to exactly approximate another one. Recall that $\widehat{M}$ approximates M when initial states and transitions in M have corresponding initial states and transitions in $\widehat{M}$. For exact approximation, we must have a type of converse as well: if $\widehat{s}$ is an initial state of $\widehat{M}$, then all of the states s of M that map to $\widehat{s}$ should be initial as well (and similarly for transitions).

Definition 5. Let $\widehat{M}$ be a transition system over $\widehat{D}$. We say that $\widehat{M}$ *exactly approximates* M (denoted $M \approx_h \widehat{M}$) when $M \sqsubseteq_h \widehat{M}$ and:

(1) $\widehat{I}(\widehat{d})$ implies $\forall d\,(h(d) = \widehat{d} \rightarrow I(d))$.

(2) $\widehat{R}(\widehat{d_1}, \widehat{d_2})$ implies $\forall d_1 \forall d_2\,(h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \rightarrow R(d_1, d_2))$.

THEOREM 4. *If $\sim$ is a congruence with respect to the primitive relations, then $M \approx_h \widehat{M}_{\min}$ (and hence $M \approx_h \widehat{M}_{\mathrm{app}}$).*

5. 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* [10] 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 $v_i = d_i$ and $v_i \neq d_i$ are atomic state formulas. Atomic state formulas are used to describe the values of variables in a state.

(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. These state formulas express that all paths (execution sequences) or some path starting at a state satisfy the property given by ϕ.

(4) If ϕ is a state formula, then ϕ is also a path formula. In this case, ϕ describes a property of the first state on the path.

(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 the following:
  (a) $\mathbf{X}\,\phi$. A path satisfies $\mathbf{X}\,\phi$ ("next time ϕ") when ϕ holds starting at the second state on the path.
  (b) $\phi\,\mathbf{U}\,\psi$. A path satisfies $\phi\,\mathbf{U}\,\psi$ ("ϕ until ψ") when ψ is true starting at some point on the path, and ϕ holds up until that point.
  (c) $\phi\,\mathbf{V}\,\psi$. The $\mathbf{V}$ operator is slightly unusual; it is the dual of $\mathbf{U}$. $\phi\,\mathbf{V}\,\psi$ is read as "ϕ releases ψ," and means that the formula ψ is true initially, and that ψ must remain true until (and including) the first point where ϕ becomes true. There is no obligation that ϕ ever become true: $\phi\,\mathbf{V}\,\psi$ also holds if ψ remains true forever.

We also use the following abbreviations: $\mathbf{F}\,\phi$ ("ϕ holds at some point in the future on the path") and $\mathbf{G}\,\phi$ ("ϕ holds globally on the path"), where ϕ is a path formula, denote $(\mathit{true}\,\mathbf{U}\,\phi)$ and $(\mathit{false}\,\mathbf{V}\,\phi)$ respectively. When specifying abstract transition systems, the atomic state formulas will take the form $\widehat{v_i} = \widehat{d_i}$ instead of $v_i = d_i$.

CTL is a restricted subset of CTL* in which the $\forall$ and $\exists$ path quantifiers may only precede a restricted set of path formulas. More precisely, CTL is the logic obtained by eliminating rules 3 through 6 above and adding the following rule.

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

CTL is of interest because there is a very efficient model checking algorithm for it [10]. $\forall$CTL* and $\forall$CTL [22; 26; 34] are restricted subsets of CTL* and CTL respectively in which the only path quantifier allowed is $\forall$. 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 a concrete transition system M over D.

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

The notation π^n will denote the suffix of π which begins at s_n. If $\pi = s_0 s_1 \ldots$ is a sequence of states from D, we denote the sequence $h(s_0)h(s_1)\ldots$ by $h(\pi)$.

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) $s \models \mathit{true}$, and $s \not\models \mathit{false}$. If $s = (e_1, \ldots, e_n)$, then $s \models v_i = d_i$ if and only if $e_i = d_i$. $s \models v_i \neq d_i$ if and only if it is not the case that $s \models v_i = d_i$.

(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) (a) $\pi \models \mathbf{X}\,\phi$ if and only if $\pi^1 \models \phi$.
 (b) $\pi \models \phi\,\mathbf{U}\,\psi$ if and only if there exists $n \in \mathcal{N}$ such that $\pi^n \models \psi$ and for all $i < n$, $\pi^i \models \phi$.
 (c) $\pi \models \phi\,\mathbf{V}\,\psi$ if and only if for all $n \in \mathcal{N}$, 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.

In the case of an abstract transition system $\widehat{M}$, we define satisfaction in exactly the same way except that the atomic formula $\widehat{v_i} = \widehat{d_i}$ is true at state $(\widehat{e_1}, \ldots, \widehat{e_n})$ if and only if $\widehat{e_i} = \widehat{d_i}$.

We now define a translation $\mathcal{C}$ between formulas describing the abstract transition $\widehat{M}$ and formulas describing M. Our goal is to be able to check a formula φ on $\widehat{M}$ and infer that the corresponding formula $\mathcal{C}(\varphi)$ holds for M. Suppose that φ is a simple atomic formula $\widehat{v_i} = \widehat{d_i}$. When this formula holds, it conceptually means that h_i applied to the value of v_i gives $\widehat{d_i}$. The only thing that we can infer at the concrete level is that $v_i = d_i$ for some d_i satisfying $h_i(d_i) = \widehat{d_i}$. Hence, $\mathcal{C}$ should map the formula $\widehat{v_i} = \widehat{d_i}$ to

$$\bigvee \{\, v_i = d_i \mid h_i(d_i) = \widehat{d_i} \,\},$$

i.e., the disjunction of all atomic formulas $v_i = d_i$ for which d_i maps to $\widehat{d_i}$. For more complex formulas, the mapping is defined recursively.

Definition 9. $\mathcal{C}$ is the mapping from formulas describing $\widehat{M}$ to formulas describing M that is defined as follows:

(1) $\mathcal{C}(\mathit{true}) = \mathit{true}$. $\mathcal{C}(\mathit{false}) = \mathit{false}$.
$\mathcal{C}(\widehat{v_i} = \widehat{d_i})$ is $\bigvee\{\, v_i = d_i \mid h_i(d_i) = \widehat{d_i} \,\}$.
$\mathcal{C}(\widehat{v_i} \neq \widehat{d_i}) = \neg\mathcal{C}(\widehat{v_i} = \widehat{d_i})$.
(2) If ϕ and ψ are state formulas, then $\mathcal{C}(\phi \wedge \psi) = \mathcal{C}(\phi) \wedge \mathcal{C}(\psi)$ and $\mathcal{C}(\phi \vee \psi) = \mathcal{C}(\phi) \vee \mathcal{C}(\psi)$.
(3) If ϕ is a path formula, then $\mathcal{C}(\forall(\phi)) = \forall(\mathcal{C}(\phi))$ and $\mathcal{C}(\exists(\phi)) = \exists(\mathcal{C}(\phi))$.
(4) If ϕ is a path formula that is also a state formula, then $\mathcal{C}(\phi)$ is given by the above rules.
(5) If ϕ and ψ are path formulas, then $\mathcal{C}(\phi \wedge \psi) = \mathcal{C}(\phi) \wedge \mathcal{C}(\psi)$ and $\mathcal{C}(\phi \vee \psi) = \mathcal{C}(\phi) \vee \mathcal{C}(\psi)$.
(6) If ϕ and ψ are path formulas, then
(a) $\mathcal{C}(\mathbf{X}\,\phi) = \mathbf{X}\,\mathcal{C}(\phi)$.
(b) $\mathcal{C}(\phi\,\mathbf{U}\,\psi) = \mathcal{C}(\phi)\,\mathbf{U}\,\mathcal{C}(\psi)$.
(c) $\mathcal{C}(\phi\,\mathbf{V}\,\psi) = \mathcal{C}(\phi)\,\mathbf{V}\,\mathcal{C}(\psi)$.

We now turn to the main theorems. For the remainder of the section, M and $\widehat{M}$ will be transition systems over D and $\widehat{D}$ respectively. First, we have a straightforward lemma that says that paths in the concrete system M can be lifted to the abstract level.

LEMMA 2. *Assume $M \sqsubseteq_h \widehat{M}$. If π is a path in M, then $h(\pi)$ is a path in $\widehat{M}$.*

Using this observation, we prove the main preservation theorem: formulas that hold at the abstract level also hold for the concrete system.

THEOREM 5. *Assume $M \sqsubseteq_h \widehat{M}$. Then:*

(1) for all $\forall CTL^*$ state formulas ϕ describing $\widehat{M}$ and every state s of M, $h(s) \models \phi$ implies $s \models \mathcal{C}(\phi)$; and
(2) for all $\forall CTL^*$ path formulas ϕ describing $\widehat{M}$ and every path π in M, $h(\pi) \models \phi$ implies $\pi \models \mathcal{C}(\phi)$.

PROOF. The proof proceeds by induction on the structure of the formula. Let $s = (e_1, \ldots, e_n)$ and $h(s) = (\widehat{e_1}, \ldots, \widehat{e_n})$.

(1) If $\phi = \mathit{true}$ or $\phi = \mathit{false}$, the result is trivial. If $\phi = (\widehat{v_i} = \widehat{d_i})$, then $h(s) \models \phi$ if and only if $\widehat{e_i} = \widehat{d_i}$. Obviously $s \models v_i = e_i$. Since we have $h_i(e_i) = \widehat{d_i}$, we can infer that s satisfies

$$\bigvee\{\, v_i = d_i \mid h_i(d_i) = \widehat{d_i} \,\}.$$

But this is just $\mathcal{C}(\widehat{v_i} = \widehat{d_i})$, and so $s \models \mathcal{C}(\widehat{v_i} = \widehat{d_i})$. The case for $\phi = (\widehat{v_i} \neq \widehat{d_i})$ is similar.
(2) $h(s) \models \phi \wedge \psi$ implies $h(s) \models \phi$ and $h(s) \models \psi$. The induction hypothesis implies $s \models \mathcal{C}(\phi)$ and $s \models \mathcal{C}(\psi)$, so $s \models \mathcal{C}(\phi \wedge \psi)$. The case for $\phi \vee \psi$ is similar.
(3) Assume $h(s) \models \forall(\phi)$. $s \models \mathcal{C}(\forall(\phi))$ if for every path π from s, $\pi \models \mathcal{C}(\phi)$. By the previous lemma, $h(\pi)$ is a path in $\widehat{M}$ from $h(s)$. Since $h(s) \models \forall(\phi)$, $h(\pi) \models \phi$. Then the induction hypothesis implies $\pi \models \mathcal{C}(\phi)$.

(4) Assume ϕ is a state formula and $h(\pi) \models \phi$. If the initial state of π is s, then the initial state of $h(\pi)$ is $h(s)$. This implies $h(s) \models \phi$, and then by the induction hypothesis, $s \models \mathcal{C}(\phi)$. Hence $\pi \models \mathcal{C}(\phi)$.

(5) The cases for the conjunction and disjunction of path formulas are similar to case 2.

(6) (a) $h(\pi) \models \mathbf{X}\,\phi$ implies $(h(\pi))^1 \models \phi$. Now $(h(\pi))^1 = h(\pi^1)$, and so the induction hypothesis implies $\pi^1 \models \mathcal{C}(\phi)$. Thus $\pi \models \mathbf{X}\,\mathcal{C}(\phi)$, and so $\pi \models \mathcal{C}(\mathbf{X}\,\phi)$.

(b) If $h(\pi) \models \phi\,\mathbf{U}\,\psi$, then there exists $n \in \mathcal{N}$ such that $(h(\pi))^n \models \psi$ and for all $i < n$, $(h(\pi))^i \models \phi$. This implies $h(\pi^n) \models \psi$ and $h(\pi^i) \models \phi$ for all $i < n$. Using the inductive hypothesis, we find $\pi \models \mathcal{C}(\phi\,\mathbf{U}\,\psi)$.

(c) The case when $h(\pi) \models \phi\,\mathbf{V}\,\psi$ is similar to the previous two cases. $\square$

COROLLARY 1. *Assume $M \sqsubseteq_h \widehat{M}$, and let ϕ be a $\forall$CTL* formula describing $\widehat{M}$. Then $\widehat{M} \models \phi$ implies $M \models \mathcal{C}(\phi)$.*

Note that this result only talks about preserving the truth of formulas that describe behavior that should hold on *all paths* from a state. Sinc the abstraction process adds extra behaviors to the model, properties describing the *existence* of a path may *not* be preserved in the same manner. Thus, verifying something like absence of deadlock at the abstract level requires proving a stronger progress property.[2]

In the case where $\widehat{M}$ exactly approximates M, we also have the converse result: satisfaction at the concrete level implies satisfaction at the abstract level. We omit the proofs here. First, we note that paths at the abstract level and at the concrete level exactly coincide.

LEMMA 3. *Assume $M \approx_h \widehat{M}$, and let π be an infinite sequence of states from S (the set of states of M). Then π is a path in M if and only if $h(\pi)$ is a path in $\widehat{M}$.*

Then we have the analog of theorem 5, except now going both ways.

THEOREM 6. *Assume $M \approx_h \widehat{M}$; then*

(1) for all CTL state formulas ϕ describing $\widehat{M}$ and every state s of M, $h(s) \models \phi$ if and only if $s \models \mathcal{C}(\phi)$; and*

(2) for all CTL path formulas ϕ describing $\widehat{M}$ and every path π in M, $h(\pi) \models \phi$ if and only if $\pi \models \mathcal{C}(\phi)$.*

COROLLARY 2. *Assume $M \approx_h \widehat{M}$, and let ϕ be a CTL* formula describing $\widehat{M}$. Then $M \models \mathcal{C}(\phi)$ if and only if $\widehat{M} \models \phi$.*

6. EXAMPLES

In this section, we discuss some abstractions which have proved useful in practice. Each is illustrated with a small example. All of the programs for the examples are given in a simple finite-state language which we now describe. Our verification system consists of a compiler for this language, plus a BDD-based model checker. Both the compiler and the model checker are written in LISP, except for the BDD routines, which are written in C.

[2]It is the opinion of one of the authors that this is what you really want to do anyway. Said author prefers systems that will do something useful to those that might...

6.1 A simple language

The language that we will be using is a procedural language designed for specifying reactive programs. The main features of this language are:

(1) It contains a variety of structured programming constructs, such as **while** loops. Non-recursive procedures are also available.

(2) It is finite state. The user must specify a fixed number of bits for each input and output in a program.

(3) 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 (i.e., occurring in zero time). There is one special statement, **wait**, which is used to indicate the passage of time. When a **wait** statement is encountered, any 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 (in zero time) until a **wait** is encountered, make output changes visible, obtain new inputs, etc. The **wait** statements indicate the control points in the program.

Aside from the **wait** statement, most of the language features used in the examples in this paper are self-explanatory.

A program in the language may be compiled into a Moore machine for verification or for implementation in hardware. Here, we are only concerned with the first of these. 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 description of the Moore machine in the form of a BDD. This is then used as the input to the BDD-based model checking program.

When a program is compiled, the user may also specify abstractions for some of the inputs or outputs. By using the techniques described previously, the compiler can directly generate an (approximate) abstract Moore machine. There are a number of abstractions built into the compiler, some of which are described in the following subsections. In addition, the user may define new abstractions by supplying procedures to build the BDDs representing them. Abstract versions of the primitive relations are computed automatically by the compiler.

Figure 3 is a small example program, a settable countdown timer, written in the language. 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.

6.2 The model checker

The model checker is essentially a propositional CTL model checker (as described by Burch *et al.* [7]), extended with a notion of types. While state components need not be only boolean, but they are restricted to finite domains. The model checker knows about all of the types allowed by the compiler. Integers are handled via two's-complement representation. When we write temporal logic formulas in this

```
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
```

Fig. 3. An example program

section, we will often write them so as to maximize readability. However, they do not necessarily represent the input format accepted by the model checker. We do this especially with abstracted variables. For example, if x is a variable that is abstracted by:

$$h(x) = \begin{cases} 0, & \text{if } x \text{ is even;} \\ 1, & \text{if } x \text{ is odd,} \end{cases}$$

then we will generally write something like $\mathrm{even}(x)$ in a formula rather than $\widehat{x} = 0$. We emphasize however that all of the properties can be expressed concisely at the abstract level when using the abstractions being considered.

6.3 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.

$$\begin{aligned} \big((i \bmod m) + (j \bmod m)\big) \bmod m &\equiv i + j \pmod m \\ \big((i \bmod m) - (j \bmod m)\big) \bmod m &\equiv i - j \pmod m \\ \big((i \bmod m)(j \bmod m)\big) \bmod m &\equiv ij \pmod m \end{aligned}$$

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.[3]

[3]It may not be immediately clear how complex representing a relationship like $i \equiv 3 \pmod 7$ is, so we briefly describe this BDD here. Suppose i is $k+1$ bits wide. If the high order (kth) bit of i is 0, then the low k bits must represent a number which is also equivalent to 3 (modulo 7).

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

THEOREM CHINESE REMAINDER THEOREM. *Let m_1, m_2, …, m_n be positive integers which are pairwise relatively prime. Define $m = m_1 m_2 \ldots m_n$, and let b, i_1, i_2, …, i_n be integers. Then there is a unique integer i such that*

$$b \le i \le b + m \qquad \text{and} \qquad i \equiv i_j \pmod{m_j} \qquad \text{for } 1 \le j \le n.$$

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, …, 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 4). 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 (bit 0) 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.[4]

The specification we used for the multiplier was a series of formulas of the following form.[5]

$$\begin{aligned}&\forall\mathbf{G}\big(\text{waiting} \land \text{req} \land (\text{in1} \bmod m = i) \land (\text{in2} \bmod m = j)\\ &\qquad \rightarrow \forall(\neg\text{ack}\ \mathbf{U}\ \text{ack} \land (\text{overflow} \lor (\text{output} \bmod m = ij \bmod m)))\big)\end{aligned}$$

Here, i and j range from 0 through $m-1$ (hence we have to check $O(m^2)$ formulas), and waiting 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

Otherwise, then the low order k bits must represent a number that is equivalent to $3 - 2^k$ (modulo 7). Both of these relationships have the same form as the original one, but they involve a number with only k bits. Further, there are only 7 modulo values that we will ever have to consider. By continuing to decompose the relationships in this way, we see that the BDD will have $O(mk)$ nodes. We also note that this is independent of the BDD variable order.

[4] 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 (overflow, output) := (output:17) + 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.

[5] 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 6.4).

```
input in1 : 16
input in2 : 16
input req : 1
output factor1 : 16 := 0
output factor2 : 16 := 0
output output : 16 := 0
output overflow : 1 := 0
output ack : 1 := 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 lsb(factor1) = 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
```

Fig. 4. A 16 bit multiplier

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.

6.4 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 4. 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.

$$\forall \mathbf{G}\big(\mathrm{waiting} \wedge \mathrm{req} \wedge (\lg \mathrm{in1} + \lg \mathrm{in2} \leq 16) \rightarrow \forall(\neg \mathrm{ack}\ \mathbf{U}\ \mathrm{ack} \wedge \neg \mathrm{overflow})\big)$$
$$\forall \mathbf{G}\big(\mathrm{waiting} \wedge \mathrm{req} \wedge (\lg \mathrm{in1} + \lg \mathrm{in2} \geq 18) \rightarrow \forall(\neg \mathrm{ack}\ \mathbf{U}\ \mathrm{ack} \wedge \mathrm{overflow})\big)$$

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.

6.5 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

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

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.

As an example of using these types of abstractions, consider the program shown in figure 5. 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

```
input in : 16
output parity : 1 := 0
output b : 16 := 0
output done : 1 := 0
b := in
wait
while b ≠ 0
    parity := parity ⊕ lsb(b)
    b := b ≫ 1
    wait
endwhile
done := 1
```

Fig. 5. A parity computation program

```
input a : 8
output b : 8 := 0
loop
    b := a
    wait
endloop
```

Fig. 6. A simple program

(2) $\sharp\mathrm{b} \oplus \mathrm{parity}$ is invariant from that point onwards.

We can express the above with the following formula.

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

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.

6.6 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. As a simple example, consider the program in figure 6.

We wish to show that the next state value of b is always equal to the current state value of a. We can express this property for a fixed value, say 42, using the formula:

$$\forall\mathbf{G}(a = 42 \rightarrow \forall\mathbf{X}\, b = 42).$$

If we wanted to verify just this property, we could use the following abstraction for a and b

$$h(i) = \begin{cases} 0, & \text{if } i = 42; \\ 1, & \text{otherwise.} \end{cases}$$

When we apply this abstraction and compile the program, we obtain the transition relation $\widehat{R}(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}')$ defined by $\widehat{b}' = \widehat{a}$. Here, the primes denote next-state variables, and all of the variables range over $\{0, 1\}$. Now to check that our program works correctly for the value 42, we would check the following formula at the abstract level:

$$\forall\mathbf{G}(\widehat{a} = 0 \rightarrow \forall\mathbf{X}\,\widehat{b} = 0).$$

The formula would of course turn out to be satisfied. Obviously though, we do not want to have to repeat this process for each possible data value.

Suppose now that we were to modify our abstraction function as follows:

$$h_c(i) = \begin{cases} 0, & \text{if } i = c; \\ 1, & \text{otherwise.} \end{cases}$$

We have introduced a new symbolic parameter that our abstraction depends on. Imagine compiling the program with this abstraction; we should get a relation $\widehat{R}_c(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}', c)$ that is parameterized by c. Fixing $c = 42$ will give the relation $\widehat{R}$ that we encountered above. If we could run the model checking algorithm on our parameterized relation, we would obtain a parameterized state set representing the states for which our formula is true. Now our specification

$$\forall\mathbf{G}(\widehat{a} = 0 \rightarrow \forall\mathbf{X}\,\widehat{b} = 0)$$

is essentially saying

$$\forall\mathbf{G}(a = c \rightarrow \forall\mathbf{X}\,b = c).$$

If the formula turns out to be true for all values of c, we will have proved the desired specification. The observation now is that by introducing 8 extra BDD variables to encode the possible choices for c, we can in fact:

(1) represent h_c with a BDD (the user will supply just h_c);

(2) compile with h_c to get a BDD representing $\widehat{R}_c(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}', c)$ (the compiler handles this step automatically);

(3) perform the model checking to obtain a BDD representing the parameterized state set (the model checker does this automatically; it simply views c as an additional state component that never changes); and

(4) if necessary, choose a specific c and generate a counterexample (also done by the model checker).

Further note that, in this case, the program behaves identically regardless of the value of c, so when we compile it, the BDD for $\widehat{R}_c$ will be independent of the extra variables that we introduced. As a result, doing the model checking will be no more complex than in the case when we were just verifying

$$\forall\mathbf{G}(a = 42 \rightarrow \forall\mathbf{X}\,b = 42).$$

In general, we have found that sharing in the BDDs makes it possible to perform efficiently the abstraction, compilation, and model checking. We call abstractions such as h_c "symbolic abstractions"; below, we give some more complex examples that make use of these abstractions.

Consider a simple partitioning:

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

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. As an example of using this abstraction, consider the program of figure 7. 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.[6]

The properties which we verified are:

(1) for every c, eventually the values of the x outputs are such that all numbers which are less than c come before all numbers which are greater than or equal to c, and this condition holds invariantly from that point on; and

(2) for every c, the number of the x outputs which are less than c 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

$$\forall\mathbf{F}\,\forall\mathbf{G}\big((\mathrm{x}_1 < c \vee \mathrm{x}_2 \geq c) \wedge \cdots \wedge (\mathrm{x}_7 < c \vee \mathrm{x}_8 \geq c)\big)$$

and

$$\big(\textstyle\sum_{i=1}^{8}(\mathrm{x}_i < c) = n\big) \rightarrow \forall\mathbf{G}\Big(\big(\textstyle\sum_{i=1}^{8}(\mathrm{x}_i < c) = n\big) \vee \neg\mathrm{stable}\Big).$$

Here, x_i is the value of the variable x in the ith cell of the array. The summation notation denotes the number of formulas $\mathrm{x}_i < c$ which are true, and stable is an atomic proposition which is true when every cell is executing the statement labeled 1.[7] Verifying these properties required just under five minutes of CPU

[6]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.

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

```
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
```

Fig. 7. A sorting cell program

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 8 and is described in detail elsewhere [6; 7]. It performs three-address arithmetic and logical operations on operands stored in a register file.

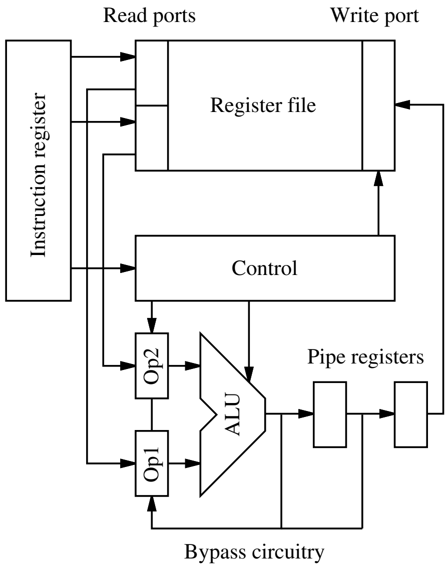

Fig. 8. Pipeline circuit block diagram

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. 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

$$\begin{aligned}&\forall\mathbf{G}\big((\mathrm{srcaddr1} = \mathrm{ra}) \wedge (\mathrm{srcaddr2} = \mathrm{rb}) \wedge (\mathrm{destaddr} = \mathrm{rc}) \wedge \neg\mathrm{stall}\\ &\qquad \rightarrow \forall\mathbf{X}\,\forall\mathbf{X}((\mathrm{regra} = \mathrm{ca}) \wedge (\mathrm{regrb} = \mathrm{cb}) \rightarrow \forall\mathbf{X}(\mathrm{regrc} = \mathrm{ca} + \mathrm{cb}))\big).\end{aligned}$$

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 over 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.* [6] 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 cubicly in the number of registers. We also note that the complexity of verifying systems like this can be further reduced using a technique that we call *symbolic compositions*. Symbolic compositions have the same flavor as symbolic abstractions, but are designed to take advantage of the compositional verification properties of ∀CTL* [22]. By combining symbolic compositions with symbolic abstractions, we were able to verify the system with 64 registers, each 64 bits, in less than 25 minutes of CPU time on a Sun 3/60, and with verification times that scale polylogarithmically in the number of registers and linearly in the width of registers. We discuss these techniques in more detail elsewhere [30].

7. 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 state programs, using techniques similar to those involved in abstract interpretation. The process 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 ∀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.

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, abstract interpretation, 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] ACM/IEEE. *Proceedings of the 28th Design Automation Conference*. IEEE Computer Society Press, June 1991.

[2] D. L. Beatty, R. E. Bryant, and C.-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In DAC91 [1], pages 397–402.

[3] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth Workshop on Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, July 1992.

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

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In DAC91 [1], pages 403–407.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th Design Automation Conference*, pages 46–51. IEEE Computer Society Press, June 1990.

[8] 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 *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January 1983.

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

[11] E. M. Clarke and S. Kimura. A parallel algorithm for constructing binary decision diagrams. In *Proceedings of the 1990 IEEE International Conference on Computer Design*, pages 220–223. IEEE Computer Society Press, October 1990.

[12] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, June 1989.

[13] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1990.

[14] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In ICCAD90 [25].

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[16] P. Cousot and R. Cousot. Systematic design of program analysis frameworks.

In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, January 1979.

[17] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1989.

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

[19] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics 19 (Mathematical Aspects of Computer Science).* American Mathematical Society, 1967.

[20] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the 1988 International Conference on Computer-Aided Design*, pages 2–5. IEEE Computer Society Press, November 1988.

[21] S. Graf and B. Steffen. Compositional minimization of finite state processes. In Kurshan and Clarke [28].

[22] O. Grumberg and D. E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, August 1991.

[23] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier, 1990.

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

[25] IEEE/ACM. *Proceedings of the 1990 International Conference on Computer-Aided Design.* IEEE Computer Society Press, November 1990.

[26] B. Josko. Verifying the correctness of AADL-modules using model checking. In de Bakker et al. [17], pages 386–400.

[27] R. P. Kurshan. Analysis of discrete event coordination. In de Bakker et al. [17], pages 414–453.

[28] R. P. Kurshan and E. M. Clarke, editors. *Proceedings of the 1990 Workshop on Computer-Aided Verification.* Springer-Verlag, June 1990.

[29] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

[30] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.

[31] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[32] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

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

[34] G. Shurek and O. Grumberg. The modular framework of computer-aided verification: Motivation, solutions and evaluation criteria. In Kurshan and Clarke [28], pages 214–223.

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

[36] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In ICCAD90 [25], pages 130–133.

[37] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986.

I don't know the received dates.