Model Checking *

E. Clarke[1], O. Grumberg[2], and D. Long[3]

[1] Carnegie Mellon, Pittsburgh
[2] The Technion, Haifa
[3] AT&T Bell Labs, Murray Hill

ABSTRACT: Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications.

This paper describes the basic model checking algorithm and shows how it can be used with binary decision diagrams to verify properties of large state-transition graphs. Abstraction and compositional reasoning techniques are also discussed that significantly extend the power of model checking techniques by exploiting the hierarchical structure of complex circuit designs and protocols.

Keywords: automatic verification, temporal logic, model checking, binary decision diagrams

Table of Contents

* This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330, and in part by the National Science foundation under Grant No. CCR-9217549 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294. 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.

1 Introduction

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine automatically if the specifications are satisfied by the state-transition graph. The technique was originally developed in 1981 by Clarke and Allen Emerson [18, 19]. Quielle and Sifakis [43] independently discovered a similar verification technique shortly thereafter. An alternative approach based on showing inclusion between ω-automata was later devised by Robert Kurshan at ATT Bell Laboratories [30, 32].

This technique has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols. The most important is that the procedure is highly automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checker will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex reactive systems.

The first model checkers were able to find subtle errors in small circuits and protocols ([6], [7], [8], [9], [19], [27], [36]). However, they were unable to handle very large examples due to the *state explosion problem*. Because of this limitation, many researchers in formal verification predicted that model checking would never be useful in practice.

The possibility of verifying systems with realistic complexity changed dramatically in the late 1980's with the discovery of how to represent transition relations using *ordered binary decision diagrams (OBDDs)* [11]. This discovery was made independently by three research teams [15, 22, 41] and is basically quite simple. Assume that the behavior of

a reactive system is determined by n boolean state variables $v_1, v_2, \ldots, v_n$. Then the transition relation of the system can be expressed as a boolean formula

$$R(v_1, v_2, \ldots, v_n, v_1', v_2', \ldots, v_n')$$

where $v_1, v_2, \ldots, v_n$ represents the current state and $v_1', v_2', \ldots, v_n'$ represents the next state. By converting this formula to a BDD, a very concise representation of the transition relation may be obtained.

The original model checking algorithm, together with the new representation for transition relations, is called *symbolic model checking* [14, 15, 16]. By using this combination, it is possible to verify extremely large reactive systems. In fact, some examples with more than than 10^{120} states have been verified [13, 16]. This is possible because the number of nodes in the OBDDs that must be constructed no longer depends on the actual number of states or the size of the transition relation. Because of this breakthrough it is now possible to verify reactive systems with realistic complexity, and a number of major companies including Intel, Motorola, Fujitsu, and ATT have started using symbolic model checkers to verify actual circuits and protocols. In several cases errors have been found that were missed by extensive simulation.

While symbolic representations have greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. In this paper we discuss two such techniques: compositional reasoning and abstraction.

The first technique exploits the *modular structure* of complex circuits and protocols. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. If it is possible to show that the system satisfies each local property, and if the conjunction of the local properties implies the overall specification, then the complete system must satisfy this specification as well [29, 34].

For instance, consider the problem of verifying a communications protocol that is modeled by three finite state processes: a transmitter, some type of network, and a receiver. Suppose that the specification for the system is that data is eventually transmitted correctly from the sender to the receiver. Such a specification might be decomposed into three local properties. First, the data should eventually be transferred correctly from the transmitter to the network. Second, the data should eventually be transferred correctly from one end of the network to the other. Finally, the data should eventually be transferred correctly from the network to the receiver. We might be able to verify the first of these local properties using only the transmitter and the network, the second using only the network, and the third using only the network and the receiver. By decomposing the verification in this way, we never have to compose all of the processes and therefore avoid the state explosion phenomenon.

The second technique involves using abstraction. This technique appears to be essential for reasoning about reactive systems that involve data paths. Traditionally, finite state verification methods have been used mainly for control-oriented systems. The symbolic methods make it possible to handle some systems that involve nontrivial data

manipulation, but the complexity of verification is often high. This approach is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. For example, in verifying the addition operation of a microprocessor, we might require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations *abstraction* can be used to reduce the complexity of model checking [20, 34]. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level.

Our paper is organized as follows: Section 2 describes the propositional temporal logic that is used for specifications. The properties of OBDDs that are needed to understand the paper are briefly discussed in Section 3. The next section gives the basic model checking algorithm and shows how it can be extended to handle fairness constraints. Section 5 describes various simulation relations between reactive systems that are used for both compositional reasoning and abstraction. Section 6 shows how model checking can extended to permit compositional reasoning based on the *assume-guarantee paradigm*. The use of abstraction to reduce the size of the state space that must be searched is discussed in Section 7. Several key abstractions are given and illustrated by example. The paper concludes in Section 8 with some directions for future research.

2 Computation tree logics

The computation tree logic CTL* [18, 19, 28] combines both branching-time and linear-time operators: a path quantifier, either **A** ("for all computation paths") or **E** ("for some computation paths") can prefix an assertion composed of arbitrary combinations of the usual linear-time operators **G** ("always"), **F** ("sometimes"), **X** ("nexttime"), **U** ("until"), and **V** ("unless"). The remainder of this section gives a precise description of the syntax and semantics of these logics.

There are two types of formulas in CTL* : *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\mathbf{E}(f)$ is a state formula.
- If f is a path formula, then $\mathbf{A}(f)$ is a state formula.

Two additional rules are needed to specify the syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X} f$, $f \mathbf{U} g$, and $f \mathbf{V} g$ are path formulas.

CTL* is the set of state formulas generated by the above rules.

In this paper finite-state systems are modeled by *Kripke structures* with *fairness constraints*. A Kripke structure $M = (S, S_0, AP, L, R, F)$ is a 6-tuple of the following form.

1. S is a finite set of states.
2. $S_0 \subseteq S$ is a set of initial states.
3. AP is a finite set of atomic propositions.
4. $L : S \to \mathcal{P}(AP)$ is a function that labels each state with the set of atomic propositions true in that state.
5. $R \subseteq S \times S$ is a transition relation.
6. $F \subseteq \mathcal{P}(S)$ is a set of fairness constraints given as Büchi acceptance conditions.

A path in M is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for all $i \geq 0$, $R(s_i, s_{i+1})$. Define $\inf(\pi) = \{ s \mid s = s_i \text{ for infinitely many } i \}$. A path π in M is *fair* if and only if for every $P \in F$, $\inf(\pi) \cap P \neq \emptyset$. We will use the notation π^n for the suffix of π which begins at s_n. Unless otherwise stated, all of our results apply only to *finite* Kripke structures.

If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in Kripke structure M. When the Kripke structure M is clear from context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $s \models p \quad \Leftrightarrow p \in L(s)$.
2. $s \models \neg f_1 \quad \Leftrightarrow s \not\models f_1$.
3. $s \models f_1 \wedge f_2 \quad \Leftrightarrow s \models f_1$ and $s \models f_2$.
4. $s \models f_1 \vee f_2 \quad \Leftrightarrow s \models f_1$ or $s \models f_2$.
5. $s \models \mathbf{E}(g_1) \quad \Leftrightarrow$ there exists a fair path π starting with s such that $\pi \models g_1$.

will 6. $s \models \mathbf{A}(g_1) \Leftrightarrow$ for all fair paths π starting with s, $\pi \models g_1$.

7. $\pi \models f_1 \quad \Leftrightarrow s$ is the first state of π and $s \models f_1$.
8. $\pi \models \neg g_1 \quad \Leftrightarrow \pi \not\models g_1$.
9. $\pi \models g_1 \wedge g_2 \quad \Leftrightarrow \pi \models g_1$ and $\pi \models g_2$.
10. $\pi \models g_1 \vee g_2 \quad \Leftrightarrow \pi \models g_1$ or $\pi \models g_2$.
11. $\pi \models \mathbf{X} g_1 \quad \Leftrightarrow \pi^1 \models g_1$.
12. $\pi \models g_1 \mathbf{U} g_2 \quad \Leftrightarrow$ there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k$, $\pi^j \models g_1$.
13. $\pi \models g_1 \mathbf{V} g_2 \quad \Leftrightarrow$ for every $k \geq 0$, if $\pi^j \not\models g_1$ for all $0 \leq j < k$, then $\pi^k \models g_2$.

We will normally be interested in the truth of a CTL* state formula f in a particular state or set of states. When f is true in all initial states of M, we will write $M \models f$.

The following abbreviations are used in writing CTL* formulas:

- $\mathbf{F} f \equiv \mathit{true}\ \mathbf{U} f$
- $\mathbf{G} f \equiv \neg \mathbf{F} \neg f$

CTL [3, 18] is a restricted subset of CTL* that permits only branching-time operators. Each of the linear-time operators **G**, **F**, **X**, and **U** must be immediately preceded by a

path quantifier. More precisely, CTL is the subset of CTL* that is obtained if the following rule is used to specify the syntax of path formulas.

- If f and g are state formulas, then $\mathbf{X}\, f$, $f\ \mathbf{U}\ g$, $f\ \mathbf{V}\ g$ are path formulas.

Each of the CTL operators can be expressed in terms of three operators **EX**, **EG**, and **EU**. For example,

- $\mathbf{AX}\, f = \neg\, \mathbf{EX}(\neg f)$
- $\mathbf{AG}\, f = \neg\, \mathbf{EF}(\neg f)$
- $\mathbf{AF}\, f = \neg\, \mathbf{EG}(\neg f)$
- $\mathbf{EF}\, f = \mathbf{E}[\mathit{true}\ \mathbf{U}\ f]$
- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\, \mathbf{E}[\neg g\ \mathbf{U}\ \neg f \wedge \neg g] \wedge \neg\, \mathbf{EG}\, \neg g$
- $\mathbf{A}[f\ \mathbf{V}\ g] = \neg\, \mathbf{E}[\neg f\ \mathbf{U}\ \neg g]$
- $\mathbf{E}[f\ \mathbf{V}\ g] = \neg\, \mathbf{A}[\neg f\ \mathbf{U}\ \neg g]$

The four operators that are used most widely are illustrated in Figure 1. Each computation tree has the state s_0 as its root.

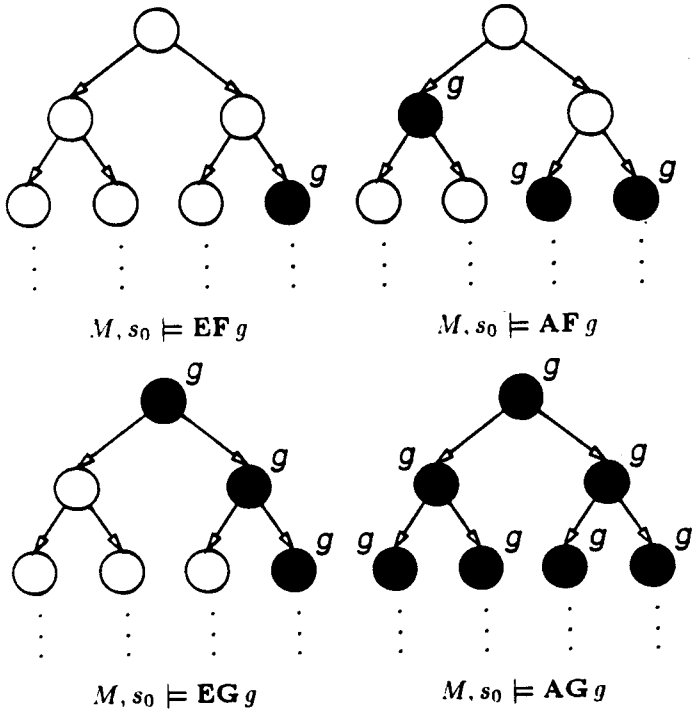

Fig. 1. Basic CTL Operators

Some typical CTL formulas that might arise in verifying a finite state concurrent program are given below:

- $\mathbf{EF}(Started \wedge \neg Ready)$: It is possible to get to a state where *Started* holds but *Ready* does not hold.
- $\mathbf{AG}(Req \rightarrow \mathbf{AF}\, Ack)$: If a request occurs, then it will be eventually acknowledged.
- $\mathbf{AG}(\mathbf{AF}\, DeviceEnabled)$: The proposition *DeviceEnabled* holds infinitely often on every computation path.
- $\mathbf{AG}(\mathbf{EF}\, Restart)$: From any state it is possible to get to the *Restart* state.
- $\mathbf{AG}(Req \rightarrow \mathbf{A}[Req\ \mathbf{U}\ Ack])$: If a request occurs, then it continues to hold, until it is eventually acknowledged. (Note that the acknowledgment must always occur.)
- $\mathbf{AG}(Req \rightarrow \mathbf{A}[Ack\ \mathbf{V}\ Req])$: Once a request occurs, then it continues to hold, until an acknowledgment occurs. (Note that the acknowledgment may never occur.)

We sometimes want to restrict the logics CTL* and CTL so that they cannot express the existence of a specific path in some Kripke structure. We do this by eliminating the existantial path quantifier from the logic. Thus, a formula may include only the universal quantifiers over paths. However, nesting of these quantifiers is allowed. To ensure that existential path quantifiers do not arise via negation, we will assume that formulas are expressed in *negation normal form*. In other words, negations are applied only to atomic propositions. The logics obtained in this manner are called *Universal CTL** (or ACTL*) and *Universal CTL* (or ACTL), respectively.

3 Binary decision diagrams

Ordered binary decision diagrams (OBDDs) are a canonical form for boolean formulas described by Bryant [11]. 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, verification of combinational logic and, more recently, verification of sequential circuits. An OBDD 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 OBDD in Figure 2. It represents the formula $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of boolean values to the variables a, b, c and d, it is possible to decide whether the assignment makes the formula true by traversing the graph beginning at the root and branching at each node based on the value assigned to the variable that labels the node. For example, the assignment $\langle a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1 \rangle$ 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 OBDD for every formula. The size of the OBDD depends critically on the variable ordering.

Most logical operations can be implemented very efficiently using OBDDs. The function that restricts some argument x_i of the boolean function f to a constant value b, denoted by $f\,|_{x_i \leftarrow b}$, can be performed in time which is linear in the size of the original binary decision diagram [11]. By using the restriction algorithm we can compute the

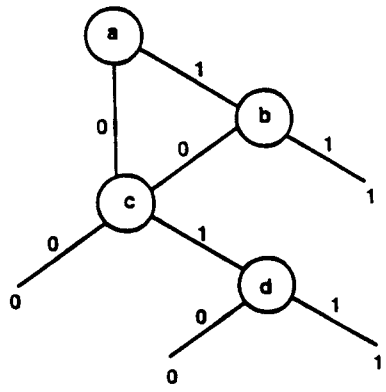

Fig. 2. OBDD for $(a \wedge b) \vee (c \wedge d)$

OBDD for the formula $\exists x f$ as $f|_{x \leftarrow 0} + f|_{x \leftarrow 1}$. All 16 two-argument logical operations can also be implemented efficiently on boolean functions that are represented as OBDDs. The complexity of these operations is linear in the size of the argument OBDDs [11]. Furthermore equivalence checking of two boolean functions can be done in constant time by using a hash table [5].

OBDDs are extremely useful for obtaining concise representations of relations over finite domains [15, 35]. If R is n-ary relation over $\{0, 1\}$ then R can be represented by the OBDD for its *characteristic function*

$$f_R(x_1, \ldots, x_n) = 1 \text{ iff } R(x_1, \ldots, x_n).$$

If R is an n-ary relation over a finite domain D, then R can be represented by an OBDD by using an appropriate binary encoding of D.

4 Model checking

Let $M = (S, S_0, AP, L, R, F)$ be an arbitrary finite Kripke structure. We use $Pred(S)$ to denote the lattice of predicates over S where each predicate is identified with the set of states in S that make it true and the ordering is set inclusion. Thus, the least element in the lattice is the empty set, denoted by $False$, and the greatest element in the lattice is the set of all states, denoted by $True$. A functional F that maps $Pred(S)$ to $Pred(S)$ will be called a *predicate transformer*. Let $\tau : Pred(S) \longrightarrow Pred(S)$ be such a functional, then

1. τ is *monotonic* provided that $P \subseteq Q$ implies $\tau[P] \subseteq \tau[Q]$;
2. τ is $\cup$-*continuous* provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau[\cup_i P_i] = \cup_i \tau[P_i]$;
3. τ is $\cap$-*continuous* provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau[\cap_i P_i] = \cap_i \tau[P_i]$.

Note that both $\cup$-continuity and $\cap$-continuity imply montonicity. Furthermore, if the domain of τ is finite, then monotonicity implies continuity.

A monotonic predicate transformer τ on $Pred(S)$ has a least fixpoint, $\mathbf{lfp}\, Z\,[\tau(Z)]$, and a greatest fixpoint, $\mathbf{gfp}\, Z\,[\tau(Z)]$ (see Tarski [44]). It is possible to show $\mathbf{lfp}\, Z\,[\tau(Z)] = \cap\{Z \mid \tau(Z) \subseteq Z\}$ if τ is monotonic and that $\mathbf{lfp}\, Z\,[\tau(Z)] = \cup_i \tau^i(False)$ if τ is also $\cup$-continuous. It is also possible to show that $\mathbf{gfp}\, Z\,[\tau(Z)] = \cup\{Z \mid \tau(Z) \supseteq Z\}$ if τ is monotonic and that $\mathbf{gfp}\, Z\,[\tau(Z)] = \cap_i \tau^i(True)$ if τ is also $\cap$-continuous.

If τ is monotonic, its least fixpoint can be computed by the program in Figure 3. The

```
function Lfp(Tau : PredicateTransformer): Predicate
begin
    Q := False;
    Q' := Tau(Q);
    while (Q ≠ Q') do
    begin
        Q := Q';
        Q' := Tau(Q')
    end;
    return(Q)
end
```

Fig. 3. Procedure for computing least fixpoints.

invariant for the while loop in the body of the procedure is given by the assertion

$$(Q' = \tau[Q]) \wedge (Q' \subseteq \mathbf{lfp}\, Z\,[\tau(Z)])$$

It is easy to see that at the beginning of the i-th iteration of the loop, $Q = \tau^{i-1}(False)$ and $Q' = \tau^i(False)$. Note, that by monotonicity,

$$False \subseteq \tau(False) \subseteq \tau^2(False) \subseteq \ldots.$$

If the loop terminates, we will have that $Q = \tau[Q]$ and that $Q \subseteq \mathbf{lfp}\, Z\,[\tau(Z)]$. It follows directly that $Q = \mathbf{lfp}\, Z\,[\tau(Z)]$ and that the value returned by the procedure is the required least fixpoint. The loop must terminate because S is finite and each iteration always increases the number of states in Q. The greatest fixpoint of τ may be computed in a similar manner by the program in Figure 4. Essentially the same argument can be used to show that the procedure terminates and that the value it returns is $\mathbf{gfp}\, Z\,[\tau(Z)]$.

If we identify each CTL formula f with the predicate $\{s \mid M, s \models f\}$ in $Pred(S)$, then each of the basic CTL operators may be characterized as a least or greatest fixpoint of an appropriate predicate transformer.

- $\mathbf{A}[f_1\ \mathbf{U}\ f_2] = \mathbf{lfp}\, Z\,[f_2 \vee (f_1 \wedge \mathbf{AX}\, Z)]$
- $\mathbf{A}[f_1\ \mathbf{V}\ f_2] = \mathbf{gfp}\, Z\,[f_2 \wedge (f_1 \vee \mathbf{AX}\, Z)]$
- $\mathbf{E}[f_1\ \mathbf{U}\ f_2] = \mathbf{lfp}\, Z\,[f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)]$
- $\mathbf{E}[f_1\ \mathbf{V}\ f_2] = \mathbf{gfp}\, Z\,[f_2 \wedge (f_1 \vee \mathbf{EX}\, Z)]$

```
function Gfp(Tau : PredicateTransformer): Predicate
begin
    Q := True;
    Q' := Tau(Q);
    while (Q ≠ Q') do
    begin
        Q := Q';
        Q' := Tau(Q')
    end;
    return(Q)
end
```

Fig. 4. Procedure for computing greatest fixpoints.

- $\mathbf{AF}\, f_1 = \mathrm{lfp}\, Z\, [f_1 \vee \mathbf{AX}\, Z]$
- $\mathbf{EF}\, f_1 = \mathrm{lfp}\, Z\, [f_1 \vee \mathbf{EX}\, Z]$
- $\mathbf{AG}\, f_1 = \mathrm{gfp}\, Z\, [f_1 \wedge \mathbf{AX}\, Z]$
- $\mathbf{EG}\, f_1 = \mathrm{gfp}\, Z\, [f_1 \wedge \mathbf{EX}\, Z]$

4.1 Symbolic model checking

Next, we describe a symbolic model checking algorithm for CTL which uses OBDDs to represent the state transition graph. Assume that the behavior of the concurrent system is determined by n boolean state variables $v_1, v_2, \ldots, v_n$. The transition relation $R(\bar{v}, \bar{v}')$ for the concurrent system will be given as a boolean formula in terms of two copies of the state variables: $\bar{v} = (v_1, \ldots, v_n)$ which represents the current state and $\bar{v}' = (v_1', \ldots, v_n')$ which represents the next state. The formula $R(\bar{v}, \bar{v}')$ is now converted to an OBDD. This usually results in a very concise representation of the transition relation.

The symbolic model checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be checked as its argument and returns an OBDD that represents exactly those states of the system that satisfy the formula. Of course, the output of *Check* depends on the system being checked; this parameter is implicit in the discussion below. We define *Check* inductively over the structure of CTL formulas. If f is an atomic proposition v_i, then $Check(f)$ is simply the OBDD for v_i. Formulas of the form $\mathbf{EX}\, f$, $\mathbf{E}[f\ \mathbf{U}\ g]$, and $\mathbf{EG}\, f$ are handled by the procedures:

$$Check(\mathbf{EX}\, f) = CheckEX(Check(f)),$$
$$Check(\mathbf{E}[f\ \mathbf{U}\ g]) = CheckEU(Check(f), Check(g)),$$
$$Check(\mathbf{EG}\, f) = CheckEG(Check(f)).$$

Notice that these intermediate procedures take OBDDs as their arguments, while *Check* takes a CTL formula as its argument. The cases of CTL formulas of the form $f \vee g$ or $\neg f$ are handled using the standard algorithms for computing boolean connectives with OBDDs. Since $\mathbf{AX}\, f$, $\mathbf{A}[f\ \mathbf{U}\ g]$ and $\mathbf{AG}\, f$ can all be rewritten using just the above operators, this definition of *Check* covers all CTL formulas.

The procedure for *CheckEX* is straightforward since the formula $\mathbf{EX}\, f$ is true in a state if the state has a successor in which f is true.

$$CheckEX(f(\bar{v})) = \exists \bar{v}' \left[f(\bar{v}') \wedge R(\bar{v}, \bar{v}') \right].$$

If we have OBDDs for f and R, then we can compute an OBDD for

$$\exists \bar{v}' \left[f(\bar{v}') \wedge R(\bar{v}, \bar{v}') \right].$$

by using the techniques described in Section 3.

The procedure for *CheckEU* is based on the least fixpoint characterization for the CTL operator **EU** that is given at the beginning of this section.

$$CheckEU(f(\bar{v}), g(\bar{v})) = \mathbf{lfp}\, Z(\bar{v}) \left[g(\bar{v}) \vee (f(\bar{v}) \wedge CheckEX(Z(\bar{v}))) \right].$$

In this case we use the function Lfp to compute a sequence of approximations

$$Q_0, Q_1, \ldots, Q_i, \ldots$$

that converges to $\mathbf{E}[f\, \mathbf{U}\, g]$ in a finite number of steps. If we have OBDDs for f, g, and the current approximation Q_i, then we can compute an OBDD for the next approximation Q_{i+1}. Since OBDDs provide a canonical form of boolean functions, it is easy to test for convergence by comparing consecutive approximations. When $Q_i = Q_{i+1}$, the function Lfp terminates. The set of states corresponding to $\mathbf{E}[f\, \mathbf{U}\, g]$ will be represented by the OBDD for Q_i.

CheckEG is similar. In this case the procedure is based on the greatest fixpoint characterization for the CTL operator **EG**.

$$CheckEG(f(\bar{v})) = \mathbf{gfp}\, Z(\bar{v}) \left[f(\bar{v}) \wedge CheckEX(Z(\bar{v})) \right].$$

Given a OBDD for f, the function Gfp can be used to compute an OBDD representation for the set of states that satisfy $\mathbf{EG}\, f$.

4.2 Fairness constraints

In the remainder of this section we describe how to modify model checking algorithm above to handle *fairness constraints*. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The path quantifiers in CTL formulas are then restricted to fair paths. We assume the fairness constraints are given by a set of CTL formulas $H = \{h_1, \ldots, h_n\}$. We define a new procedure *CheckFair* for checking CTL formulas relative to the fairness constraints in H. We do this by giving definitions for new intermediate procedures *CheckFairEX*, *CheckFairEU*, and *CheckFairEG* which correspond to the intermediate procedures used to define *Check*.

Consider the formula $\mathbf{EG}\, f$ given fairness constraints H. The formula means that there exists a path beginning with the current state on which f holds globally (invariantly) and each formula in H holds infinitely often on the path. The set of such states S is the largest set with the following two properties:

1. all of the states in S satisfy f, and

2. for all fairness constraints $h_k \in H$ and all states $s \in S$, there is a sequence of states of length one or greater from s to a state in S satisfying h_k such that all states on the path satisfy f.

It is easy to show that if these conditions hold, each state in the set is the beginning of an infinite computation path on which f is always true, and for which every formula in H holds infinitely often. Thus, the procedure $CheckFairEG(f(\bar{v}))$ will compute the greatest fixpoint

$$\mathbf{gfp}\, Z(\bar{v})\,[f(\bar{v}) \wedge \bigwedge_{k=1}^{n} CheckEX(CheckEU(f(\bar{v}), Z(\bar{v}) \wedge Check(h_k)))].$$

The fixed point can be evaluated in the same manner as before. The main difference is that each time the above expression is evaluated, several nested fixed point computations are done (inside $CheckEU$).

Checking $\mathbf{EX}\, f$ and $\mathbf{E}[f\,\mathbf{U}\,g]$ under fairness constraints is simpler. The set of all states which are the start of some fair computation is

$$fair(\bar{v}) = CheckFairEG(True).$$

The formula $\mathbf{EX}\, f$ is true under fairness constraints in a state s if and only if there is a successor state s' such that s' satisfies f and s' is at the beginning of some fair computation path. It follows that the formula $\mathbf{EX}\, f$ (under fairness constraints) is equivalent to the formula $\mathbf{EX}(f \wedge fair)$ (without fairness constraints). Therefore, we define

$$CheckFairEX(f(\bar{v})) = CheckEX(f(\bar{v}) \wedge fair(\bar{v})).$$

Similarly, the formula $\mathbf{E}[f\,\mathbf{U}\,g]$ (under fairness constraints) is equivalent to the formula $\mathbf{E}[f\,\mathbf{U}\,(g \wedge fair)]$ (without fairness constraints). Hence, we define

$$CheckFairEU(f(\bar{v}), g(\bar{v})) = CheckEU(f(\bar{v}), g(\bar{v}) \wedge fair(\bar{v})).$$

5 Equivalences and preorders between structures

To avoid the state explosion problem, we would like to develop techniques that replace a large structure by a smaller structure that satisfies the same properties. More specifically, given a logic $\mathcal{L}$ and a structure M, we would like to find a smaller structure M' that satisfies exactly the same set of formulas of the logic $\mathcal{L}$ as M. In order to accomplish this goal, we need a notion of equivalence between structures that can be efficiently computed and guarantees that two structures satisfy the same set of formulas in $\mathcal{L}$. We first consider the logic CTL* and *bisimulation equivalence*.

Given two structures M and M' with the same set of atomic propositions AP, a relation $B \subseteq S \times S'$ is a *bisimulation relation* between M and M' if and only for all s and s', if $B(s, s')$ then the following conditions hold:

1. $L(s) = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$ there is s'_1 such that $R'(s', s'_1)$ and $B(s_1, s'_1)$.
3. For every state s'_1 such that $R'(s', s'_1)$ there is s_1 such that $R(s, s_1)$ and $B(s_1, s'_1)$.

The structures M and M' are *bisimulation equivalent* (denoted $M \equiv M'$) if there exists a bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0, s'_0)$.

The following lemma is important in establishing the connection between CTL* and bisimulation equivalence. We say that two paths $\pi = s_0 s_1, \ldots$ in M and $\pi' = s'_0 s'_1, \ldots$ in M' *correspond* if and only if for every i, $B(s_i, s'_i)$.

Lemma 1. *Let s and s' be two states such that $B(s, s')$. Then for every path starting from s there is a corresponding path starting from s', and for every path starting from s' there is a corresponding path starting from s.*

Proof. Let $B(s, s')$ and let $\pi = s_0 s_1, \ldots$ be a path from $s = s_0$. We construct a corresponding path $\pi' = s'_0 s'_1, \ldots$ from $s' = s'_0$ by induction. It is clear that $B(s_0, s'_0)$. Assume $B(s_i, s'_i)$ for some i. We will show how to choose s'_{i+1}. Since $B(s_i, s'_i)$ and $R(s_i, s_{i+1})$, there must be a successor t' of s'_i such that $B(s_{i+1}, t')$. We choose s'_{i+1} to be t'. □

The next lemma shows that if two states are bisimilar, then they satisfy the same set of CTL* state formulas. Furthermore, if two paths correspond, then they satisfy the same set of path formulas.

Lemma 2. *Let f be either a state formula or a path formula. Assume that s and s' are bisimilar states and that π and π' are corresponding paths. Then,*

- *$s \models f \Leftrightarrow s' \models f$ if f is a state formula, and*
- *$\pi \models f \Leftrightarrow \pi' \models f$ if f is a path formula.*

Proof. We prove the lemma by induction on the structure of f.
Base: $f = p$ for $p \in AP$. Since $B(s, s')$, we know that $L(s) = L'(s')$. Thus, $s \models p$ if and only if $s' \models p$.
Induction: There are several cases.

1. $f = \neg f_1$, a state formula.

$$\begin{aligned} s \models f &\Leftrightarrow s \not\models f_1 \\ &\Leftrightarrow s' \not\models f_1 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow s' \models f \end{aligned}$$

   The same reasoning holds if f is a path formula.
2. $f = f_1 \vee f_2$, a state formula.

$$\begin{aligned} s \models f &\Leftrightarrow s \models f_1 \text{ or } s \models f_2 \\ &\Leftrightarrow s' \models f_1 \text{ or } s' \models f_2 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow s' \models f \end{aligned}$$

   We can also use this argument if f is a path formula.
3. $f = f_1 \wedge f_2$, a state formula. This case is similar to the previous case. Furthermore, the same argument can be used if f is a path formula.

4. $f = \mathbf{E}\, f_1$, a state formula. Suppose that $s \models f$. Then there is a path π_1 starting from s such that $\pi_1 \models f_1$. By lemma 1, there is a corresponding path π_1' in M' starting from s'. So by the induction hypothesis, $\pi_1 \models f_1$ if and only if $\pi_1' \models f_1$. Therefore, $s' \models \mathbf{E}\, f_1$. The same argument can be used to prove that if $s' \models f$ then $s \models f$.
5. $f = \mathbf{A}\, f_1$, a state formula. The argument for this case is similar to the argument for $f = \mathbf{E}\, f_1$ and will not be given.
6. $f = f_1$, where f is a path formula and f_1 is a state formula. Although the lengths of f and f_1 are the same, we can imagine that $f = \mathbf{path}(f_1)$, where **path** is a special operator which converts a state formula into a path formula. Therefore, we are simplifying f by dropping this **path** operator. If s_0 and s_0' are the first states of π and π', respectively, then

$$\begin{aligned} \pi \models f &\Leftrightarrow s_0 \models f_1 \\ &\Leftrightarrow s_0' \models f_1 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow \pi' \models f \end{aligned}$$

7. $f = \mathbf{X}\, f_1$, a path formula. Suppose $\pi \models f$. By the definition of the next time operator, $\pi^1 \models f_1$. Since π and π' correspond, so do π^1 and π'^1. Therefore, by the induction hypothesis, $\pi'^1 \models f_1$, and so $\pi' \models f$. The same argument can be used to prove that if $\pi' \models f$ then $\pi \models f$.
8. $f = f_1\, \mathbf{U}\, f_2$, a path formula. Suppose that $\pi \models f_1\, \mathbf{U}\, f_2$. By the definition of the until operator, there is a k such that $\pi^k \models f_2$ and for all $0 \leq j < k$, $\pi^j \models f_1$. Since π and π' correspond, so do π^j and π'^j for any j. Therefore, by the induction hypothesis, $\pi'^k \models f_2$ and for all $0 \leq j < k$, $\pi'^j \models f_1$. Therefore, $\pi' \models f$. The same argument can be used to prove that if $\pi' \models f$ then $\pi \models f$.
9. $f = f_1\, \mathbf{V}\, f_2$, a path formula. The argument in this case is similar to the argument for $f = f_1\, \mathbf{U}\, f_2$ and will not be given. □

The next theorem is a consequence of the preceding lemma.

Theorem 3. *If $B(s, s')$ then for every CTL* formula f, $s \models f \Leftrightarrow s' \models f$.*

If two structures are bisimulation equivalent, then every initial state of one is bisimilar to some initial state of the other. Since a structure satisfies a formula if and only if each of its initial states satisfies the formula, both structures will satisfy the same set of CTL* formulas.

Theorem 4. *If $M \equiv M'$ then for every CTL* formula f, $M \models f \Leftrightarrow M' \models f$.*

The converse of this theorem is also true. If two structures satisfy the same set of CTL* formulas then they are bisimulation equivalent. In fact we can show that if two structures satisfy the same CTL formulas they are bisimulation equivalent. It follows that if two structures can be ***distinguished*** by a formula of CTL*(i.e., there is a CTL* formula that is true of one structure and not of the other) then they can also be distinguished by a formula of CTL. These results are described in [10].

The notion of bisimulation equivalence can be extended to structures with *fairness constraints*. Let M and M' be two structures with fairness constraints. Assume that

both have the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a *fair bisimulation relation* between M and M' if and only if for all s and s', if $B(s, s')$ the following conditions hold.

1. $L(s) = L'(s')$.
2. For every *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M there is a *fair* path $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' such that for all $i \geq 0$, $B(s_i, s'_i)$.
3. For every *fair* path $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' there is a *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M such that for all $i \geq 0$, $B(s_i, s'_i)$.

In this case, two structures M and M' are *fair bisimulation equivalent* (denoted $M \equiv_F M'$) if there exists a fair bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0, s'_0)$. If the semantics of CTL* is given with respect to fair paths then we can prove an analog of Theorem 4 for fair structures.

Theorem 5. *If $M \equiv_F M'$, then for every CTL* formula f interpreted over fair paths, $M \models f \Leftrightarrow M' \models f$.*

The proof of this theorem is similar to the proof of the previous theorem and is omitted.

Sometimes bisimulation equivalence does not result in a significant reduction in the number of states. By restricting the logic and relaxing the requirement that the structures should satisfy exactly the same formulas, a greater reduction can be obtained. In order to achieve this goal we introduce the notion of a *simulation relation*. Simulation is closely related to bisimulation. Bisimulation guarantees that two structures have the same behaviors. Simulation, on the other hand, relates a structure to an *abstraction* of the structure. It guarantees that every behavior of a structure is also a behavior of its abstraction. However, the abstraction might have behaviors that are not possible in the original structure. Since the abstraction may hide some of the details of the original structure, it may have a smaller set of atomic propositions.

Given two structures M and M' with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a *simulation relation* between M and M' if and only if for all s and s', if $H(s, s')$ then the following conditions hold.

1. $L(s) \cap AP' = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$, there is a state s'_1 with the property that $R'(s', s'_1)$ and $H(s_1, s'_1)$.

We say that M' *simulates* M (denoted by $M \preceq M'$) if there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s'_0 in M' for which $H(s_0, s'_0)$.

Lemma 6. *$\preceq$ is a preorder on the set of structures.*

Proof. The relation $H = \{ (s, s) \mid s \in S \}$ is a simulation between M to M, so $\preceq$ is reflexive. Thus it only remains to show that $\preceq$ is transitive. Assume $M \preceq M'$ and $M' \preceq$

M''. Let H_0 be a simulation between M and M', and let H_1 be a simulation between M' to M''. Define H_2 as the relational product of H_0 and H_1, i.e.,

$$H_2 = \{ (s, s'') \mid \exists s' [H_0(s, s') \wedge H_1(s', s'')] \}.$$

If $s_0 \in S_0$, then by the definition of simulation, there exists $s_0' \in S_0'$ such that $H_0(s_0, s_0')$. Similarly, there exists $s_0'' \in S_0''$ such that $H_1(s_0', s_0'')$, and hence $H_2(s_0, s_0'')$.

Suppose $H_2(s, s'')$, and let s' be such that $H_0(s, s')$ and $H_1(s', s'')$. By the definition of simulation, $L(s) \cap AP' = L'(s')$ and $L'(s') \cap AP'' = L''(s'')$. Then since $AP' \supseteq AP''$, we have $L(s) \cap AP'' = L''(s'')$. Let $R(s, s_1)$ be a transition in M from s. Then there exists a transition $R(s', s_1')$ in M' such that $H_0(s_1, s_1')$. Since H_1 is a simulation, there exists a transition $R''(s'', s_1'')$ in M'' such that $H_1(s_1', s_1'')$. Hence $H_2(s, s_1'')$, and H_2 is a simulation between M and M''. Thus $M \preceq M''$. □

The following lemma is the analog of Lemma 1 for simulation relations. In this case, we also say that paths $\pi = s_0 s_1 \ldots$ in M and $\pi' = s_0' s_1' \ldots$ in M' *correspond* if and only if for every i, $H(s_i, s_i')$.

Lemma 7. *Assume that s and s' are states such that $H(s, s')$. Then for every path π starting from s there is a corresponding path π' starting from s'.*

Theorem 8. *Suppose $M \preceq M'$. Then for every ACTL* formula f (with atomic propositions in AP'), $M' \models f$ implies $M \models f$.*

Intuitively, this theorem is true because formulas in ACTL* describe properties that are true of all behaviors of a structure. Since every behavior of M is a behavior of M', every formula of ACTL* that is true in M' must also be true in M. A formal proof can be obtained from Lemma 7 by using an argument similar to the one used to establish Theorem 4.

Simulation can be extended to fair structures in the same way that bisimulation is extended to fair structures. Let M and M' be two structures with fairness constraints. Assume that $AP \supseteq AP'$. The relation $H \subseteq S \times S'$ is a *fair simulation relation* between M and M' if and only if for all s and s', if $H(s, s')$ then the following conditions hold:

1. $L(s) \cap AP' = L'(s')$.
2. For every *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M, there is a fair *path* $\pi' = s_0' s_1' \ldots$ from $s' = s_0'$ in M' such that for all $i \geq 0$, $H(s_i, s_i')$.

We write $M \preceq_F M'$ if there exists a fair simulation relation H such that for every initial state $s_0 \in S_0$ in M there is an initial state $s_0' \in S_0'$ in M' such that $H(s_0, s_0')$. It is easy to show that $\preceq_F$ determines a preorder on fair structures. When it is clear from the context that we are dealing with fair simulation, we will sometimes use $\preceq$.

Every fair behavior of M is a fair behavior of M'. Thus, if the semantics of ACTL* is given with respect to fair paths then we can prove the following theorem.

Theorem 9. *If $M \preceq_F M'$, then for every ACTL* formula f interpreted over fair paths, $M' \models f$ implies $M \models f$.*

5.1 Algorithms for bisimulations and simulations

We next consider algorithms that determine whether two structures are bisimulation equivalent or whether one structure precedes another in the simulation preorder. Bisimulation equivalence is easy to check if both structures are *deterministic*, i.e., if $R(s,t)$ and $R(s,u)$, then $L(t) \neq L(u)$ and the initial states of each machine have distinct labels. For deterministic structures, bisimulation equivalence can be reduced to language equivalence. The *language of a structure* is the set of sequences of labelings which occur along all paths that start from initial states. It can be shown that two deterministic structures are bisimulation equivalent if and only if they have the same language. Extremely efficient algorithms are known for checking language equivalence [17]. These algorithms can be used to check bisimulation equivalence for deterministic structures.

We now present a general algorithm that handles both deterministic and nondeterministic structures that do not include fairness constraints. Let M and M' be two structures with the same set of atomic propositions AP, we define a sequence of relations $B_0^*, B_1^*, \ldots$ on $S \times S'$ as follows:

1. $B_0^*(s,s')$ if and only if $L(s) = L'(s')$;
2. $B_{n+1}^*(s,s')$ if and only if
   - $B_n^*(s,s')$;
   - $\forall s_1[R(s,s_1) \implies \exists s_1'[R'(s',s_1') \wedge B_n^*(s_1,s_1')]]$, and
   - $\forall s_1'[R'(s',s_1') \implies \exists s_1[R(s,s_1) \wedge B_n^*(s_1,s_1')]]$.

We write $B^*(s,s')$ if and only if $B_i^*(s,s')$ for all $i \geq 0$. Two structures M and M' are *B^*-equivalent* if for every initial state $s_0 \in S_0$ in M there is an initial state $s_0' \in S_0'$ in M' such that $B^*(s_0,s_0')$. In addition, for every initial state $s_0' \in S_0'$ in M' there is an initial state $s_0 \in S_0$ in M such that $B^*(s_0,s_0')$. It is easy to see that B^* is a bisimulation between M and M'. In fact, it can be shown that B^* is the *largest* such bisimulation, i.e., every bisimulation between M and M' is included in B^*.

Proposition 10. *B^* is the largest bisimulation between M and M'.*

Proof. It is sufficient to prove that if B is a bisimulation between M and M', then B is contained in B_i^* for every $i \geq 0$. We show this by induction on i. Clearly, B is contained in B_0^*, since any pair of states in B have the same labeling. Assume that B is contained in B_n^* and $B(s,s')$. Let $R(s,s_1)$ be a transition in M. Since B is a bisimulation, there exists a state s_1' such that $R'(s',s_1')$ is a transition in M' and $B(s_1,s_1')$. Since B is contained in B_n^*, we have that $B_n^*(s_1,s_1')$. The third requirement can be proved in a similar manner. Thus, $B_{n+1}^*(s,s')$. □

Since the structures are finite, there exists some n such that $B^* = B_n^*$. Thus, the definition gives an algorithm for computing the largest bisimulation between two structures. If an explicit state representation is used for the transition relations, then the algorithm is polynomial. A more efficient polynomial algorithm for this case is given in [40]. If OBDDs are used to represent the transition relations, then the definition can be used directly to compute the largest bisimulation—it just describes the computation of the greatest fixpoint of an appropriate functional.

Algorithms for checking fair bisimulation have not been widely investigated. If the structures are deterministic, then an efficient algorithm, based on language equivalence, can also be given in this case. The only change that is necessary is to restrict the language of a structure to fair paths. With this change it is possible to prove that two structures are fair bisimulation equivalent if and only if they are language equivalent with respect to fair paths. Thus, algorithms that check language equivalence for fair structures [17] can be used to handle this case. A general procedure that also handles nondeterministic structures is given in [1]; however, it is not clear if it can be implemented efficiently.

Each of the algorithms mentioned above can be adapted to check the simulation preorder between two structures M and M'. Language inclusion replaces language equivalence in the deterministic case. For the general case without fairness, we define a sequence of relations $H_0^*, H_1^*, \ldots$ on $S \times S'$ as follows:

1. $H_0^*(s, s')$ if and only if $L(s) \cap AP' = L'(s')$;
2. $H_{n+1}^*(s, s')$ if and only if
   - $H_n^*(s, s')$, and
   - $\forall s_1[R(s, s_1) \implies \exists s_1'[R'(s', s_1') \wedge H_n^*(s_1, s_1')]]$;

The procedure is guaranteed to terminate since the structures are finite. We write $H^*(s, s')$ if and only if $H_i^*(s, s')$ for all $i \geq 0$. As in the previous case, H^* is the largest simulation relation between the two structures M and M'.

6 Compositional Reasoning

Compositional reasoning exploits the natural decomposition of a complex system into simpler components. Properties of the individual components are verified first. When a component is verified it may be necessary to *assume* that the environment behaves in a certain manner. If the other components in the system *guarantee* this behavior, then we can conclude that the verified properties are true of the entire system. These properties can be used to deduce additional global properties of the system.

Pnueli's *assume-guarantee paradigm* [42] uses this method. A formula in his logic is a triple $\langle g \rangle M \langle f \rangle$ where g and f are temporal formulas and M is a program. The formula is true if whenever M is part of a system satisfying g, the system must also satisfy f. A typical proof shows that $\langle g \rangle M \langle f \rangle$ and $\langle \mathit{true} \rangle M' \langle g \rangle$ hold and concludes that $\langle \mathit{true} \rangle M \parallel M' \langle f \rangle$ is true. This proof strategy can also be expressed as an inference rule:

$$\frac{\langle g \rangle M \langle f \rangle \quad \langle \mathit{true} \rangle M' \langle g \rangle}{\langle \mathit{true} \rangle M \parallel M' \langle f \rangle}$$

To automate this type of reasoning, we use the simulation preorder and the logic ACTL. The preorder has the property that if a formula is true for a model, it is true for any model that precedes it in the preorder. We define parallel composition so that a system precedes any component that it contains in the preorder. Moreover, composition with a fixed component preserves the preorder relation between two components. Finally, we construct a special model of a formula, called a *tableau*. The *tableau* has the property that a structure satisfies a formula if and only if it precedes the tableau of the formula

in the preorder. In such a framework, the above reasoning sequence might be expressed as: $\mathcal{T}$ is the tableau of g, $M \parallel \mathcal{T} \models f$, $M' \preceq \mathcal{T}$, and hence $M \parallel M' \models f$.

6.1 Composition of structures

Let $M = (S, S_0, AP, L, R, F)$ and $M' = (S', S'_0, AP', L', R', F')$ be two structures. The *parallel composition* of M and M', denoted $M \| M'$, is the structure M'' defined as follows.

1. $S'' = \{ (s, s') \mid L(s) \cap AP' = L'(s') \cap AP \}$.
2. $S''_0 = (S_0 \times S'_0) \cap S''$.
3. $AP'' = AP \cup AP'$.
4. $L''((s, s')) = L(s) \cup L(s')$.
5. $R''((s, s'), (t, t'))$ if and only if $R(s, t)$ and $R'(s', t')$.
6. $F'' = \{ (P \times S') \cap S'' \mid P \in F \} \cup \{ (S \times P') \cap S'' \mid P' \in F' \}$.

This definition of composition models *synchronous* behavior. States of the composition are pairs of component states that agree on the common atomic propositions. Each transition of the composition involves a joint transition of the two components. The definition is relatively straightforward with the exception of the fairness constraint. The constraint is designed to insure the *fair path property* which states that a path in $M \| M'$ is fair if and only if its restriction to each component results in a fair path. Intuitively, the first set of pairs in the constraint

$$\{ (P \times S') \cap S'' \mid P \in \mathcal{F} \}$$

insures that the restriction of a path in M'' to its component in S is a fair path in M. The second set of pairs

$$\{ (S \times P') \cap S'' \mid P' \in \mathcal{F}' \}$$

insures that the restriction of the path to its component in S' is a fair path in M'. Since $P \times S'$ and $S \times P'$ may contain pairs that are not states in M'', it is necessary to intersect each with S''.

It is straightforward but tedious to prove that parallel composition is commutative and associative (up to isomorphism). The next three theorems deal with the connection between parallel composition and the simulation preorder $\preceq$. The first theorem states that composing M with M' can only restrict the possible behaviors of M. As a consequence of this theorem, it is sufficient to reason about the structure M rather than arbitrary systems containing M. Moreover, this theorem and Theorem 8 imply that a standard CTL model checker [19] can be used to determine if a formula of ACTL is true in all systems containing a given component. This is the key to compositional verification.

Theorem 11. *For all M and M', $M \| M' \preceq M$.*

Proof. Let S'' be the set of states of $M \| M'$. Define H by

$$H = \{ ((s, s'), s) \mid (s, s') \in S'' \}.$$

If (s_0, s'_0) is an initial state of $M \| M'$, then $s_0 \in S_0$. The label of (s, s') is $L(s) \cup L'(s')$, and $(L(s) \cup L'(s')) \cap AP = L(s)$. If $(s_0, s'_0)(s_1, s'_1) \ldots$ is a fair path in $M \| M'$, then by the fair path property, $s_0 s_1 \ldots$ is a fair path in M. By the definition of H, we have $H((s_i, s'_i), s_i)$ for every i. Hence H is a simulation relation and $M \| M' \preceq M$. $\square$

The second theorem permits a component of a system to be replaced by an abstraction of that component. Thus, in order to show that some property is true in the system $M \| M''$, we can replace M by an abstraction M' and then verify that the property holds in $M' \| M''$. Checking $M \preceq M'$ insures that M' is indeed an abstraction of M.

Theorem 12. *For all M, M' and M'', if $M \preceq M'$ then $M \| M'' \preceq M' \| M''$.*

Proof. Let H_0 be a simulation relation between M and M'. Define H_1 by

$$H_1 = \{ ((s, s''), (s', s'')) \mid H_0(s, s') \}.$$

We show that H_1 is a simulation relation. Let (s_0, s_0'') be an initial state of $M \| M''$. By the definition of composition, $s_0 \in S_0$ and $s_0'' \in S_0''$. Since $M \preceq M'$, there exists $s_0' \in S_0'$ such that $H_0(s_0, s_0')$. Next we show that (s_0', s_0'') is a state of $M' \| M''$. We break this task into several steps.
Since $H_0(s_0, s_0')$ we have

$$L'(s_0') = L(s_0) \cap AP'.$$

By intersecting both sides with AP'' we obtain

$$L'(s_0') \cap AP'' = (L(s_0) \cap AP') \cap AP''.$$

Using associativity and commutativity of set intersection we get

$$(L(s_0) \cap AP') \cap AP'' = (L(s_0) \cap AP'') \cap AP'.$$

Since $(s_0, s_0'') \in S''$, we have $L(s_0) \cap AP'' = L''(s_0'') \cap AP$; this implies

$$(L(s_0) \cap AP'') \cap AP' = (L''(s_0'') \cap AP) \cap AP'.$$

Since $M \preceq M'$, $AP \supseteq AP'$; thus,

$$(L''(s_0'') \cap AP) \cap AP' = L''(s_0'') \cap AP'.$$

Combining the above steps we obtain,

$$L'(s_0') \cap AP'' = L''(s_0'') \cap AP'.$$

Consequently, (s_0', s_0'') is a state of $M' \| M''$. Further, (s_0', s_0'') is an initial state of $M' \| M''$ by the definition of composition. By definition of H_1, we have $H_1((s_0, s_0''), (s_0', s_0''))$.

Suppose $H_1((s, s''), (s', s''))$. First we show that the two states (s, s'') and (s', s'') have the appropriate labeling. By distributivity it follows that

$$((L(s) \cup L''(s'')) \cap (AP' \cup AP'')$$

is the same as

$$(L(s) \cap AP') \cup (L(s) \cap AP'') \cup (L''(s'') \cap (AP' \cup AP'')).$$

Since $H_0(s, s')$, we have $L(s) \cap AP' = L'(s')$. Since $(s, s'') \in M \| M''$, we have $L(s) \cap AP'' = L''(s'') \cap AP$. Since $L''(s'') \subseteq AP''$, we have $L''(s'') \cap (AP' \cup AP'') = L''(s'')$. Thus, the previous expression is equal to

$$L'(s') \cup (L''(s'') \cap AP) \cup L''(s'').$$

Since $L''(s'') \cap AP \subseteq L''(s'')$, this expression simplifies to

$$L'(s') \cup L''(s'').$$

Consequently, we have

$$(L(s) \cup L''(s'')) \cap (AP' \cup AP'') = L'(s') \cup L''(s'').$$

Let $(s_0, s_0'')(s_1, s_1'') \ldots$ be a fair path in $M \parallel M''$ from $(s, s'') = (s_0, s_0'')$. Then for every $i \geq 0$, we have $L(s_i) \cap AP'' = L''(s_i'') \cap AP$. By the fair path property, $\pi = s_0 s_1 \ldots$ is a fair path in M starting at s, and $\pi'' = s_0'' s_1'' \ldots$ is a fair path in M'' from s''. Since $H_0(s, s')$, there is a path $\pi' = s_0' s_1' \ldots$ from $s' = s_0'$ in M' such that for every $i \geq 0$, $H_0(s_i, s_i')$. By the definition of a simulation relation, $L(s_i) \cap AP' = L'(s_i')$ for all i. Arguing as above, we then have $L'(s_i') \cap AP'' = L''(s_i'') \cap AP'$ for each i, and so each (s_i', s_i'') is a state in $M' \parallel M''$. Now $H_1((s_i, s_i''), (s_i', s_i''))$ by the definition of H_1. By the fair path property we see that $(s_0', s_0'')(s_1', s_1'') \ldots$ is a fair path starting in (s', s'') and corresponding to the path $(s_0, s_0'')(s_1, s_1'') \ldots$ □

The last theorem is a technical result that is needed in order to use multiple levels of assume-guarantee reasoning. We will see how it is used in section 6.3.

Theorem 13. *For all M, $M \preceq M \parallel M$.*

Proof. First note that for every state s of M, (s, s) is a state of $M \parallel M$. Define $H = \{ (s, (s, s)) \mid s \in S \}$. If $s_0 \in S_0$, then by the definition of composition, (s_0, s_0) is an initial state of $M \parallel M$. (s, s) trivially has the same label as s. Using the fair path property and the definition of composition, we find that if $s_0 s_1 \ldots$ is a fair path in M, then $(s_0, s_0)(s_1, s_1) \ldots$ is a fair path in $M \parallel M$. By the definition of H, we have $H(s_i, (s_i, s_i))$ for all i. Hence H is a simulation relation and $M \preceq M \parallel M$. □

6.2 Tableau construction

In this section, we give a tableau construction for ACTL formulas. A similar construction for LTL is given in [15]. Other tableau constructions for temporal logics can be found in [33, 45]. We show that the tableau of a formula is a *maximal model* for the formula under the relation $\preceq$. This is the key property of the tableau construction. The tableau generated by the construction can be used as an assumption by composing it with the given system before applying the model checking algorithm. Discharging the assumption is simply a matter of checking that the environment satisfies the formula. We also indicate how the tableau can be used to do temporal reasoning. For the remainder of this section, fix an ACTL formula f.

We now describe the construction of the tableau $\mathcal{T}_f$ for f in detail. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $\mathcal{T}_f$ with AP_f as its set of atomic propositions. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively as follows:

1. $el(p) = el(\neg p) = \{p\}$ if $p \in AP_f$.

2. $el(g_1 \vee g_2) = el(g_1 \wedge g_2) = el(g_1) \cup el(g_2)$.
3. $el(\mathbf{AX}\, g_1) = \{\mathbf{AX}\, g_1\} \cup el(g_1)$.
4. $el(\mathbf{A}[g_1\ \mathbf{U}\ g_2]) = \{\mathbf{AX}\, \mathit{false}, \mathbf{AX}(\mathbf{A}[g_1\ \mathbf{U}\ g_2])\} \cup el(g_1) \cup el(g_2)$.
5. $el(\mathbf{A}[g_1\ \mathbf{V}\ g_2]) = \{\mathbf{AX}\, \mathit{false}, \mathbf{AX}(\mathbf{A}[g_1\ \mathbf{V}\ g_2])\} \cup el(g_1) \cup el(g_2)$.

The special elementary subformula $\mathbf{AX}\, \mathit{false}$ denotes the nonexistence of a fair path: $s \models \mathbf{AX}\, \mathit{false}$ indicates that no fair path begins at s.

The set of states S_T of the tableau is $\mathcal{P}(el(f))$. The labeling function L_T is defined so that each state is labeled by the set of atomic propositions contained in the state. In order to specify the set of initial states and the transition relation R_T, we need an additional function *sat* that associates with each subformula g of f a set of states in S_T. Intuitively, $sat(g)$ will be the set of states that satisfy g.

1. $sat(g) = \{s \mid g \in s\}$ where $g \in el(f)$.
2. $sat(\neg g) = \{s \mid g \notin s\}$ where g is an atomic proposition.
3. $sat(g \vee h) = sat(g) \cup sat(h)$.
4. $sat(g \wedge h) = sat(g) \cap sat(h)$.
5. $sat(\mathbf{A}[g\ \mathbf{U}\ h]) = (sat(h) \cup (sat(g) \cap sat(\mathbf{AX}(\mathbf{A}[g\ \mathbf{U}\ h])))) \cup sat(\mathbf{AX}\, \mathit{false})$.
6. $sat(\mathbf{A}[g\ \mathbf{V}\ h]) = (sat(h) \cap (sat(g) \cup sat(\mathbf{AX}(\mathbf{A}[g\ \mathbf{V}\ h])))) \cup sat(\mathbf{AX}\, \mathit{false})$.

The set of initial states of the tableau is $S_0^T = sat(f)$. We want the transition relation to have the property that each elementary formula in a state is true of that state. Clearly, if $\mathbf{AX}\, g$ is in some state s, then all the successors of s should satisfy g. On the other hand, if $\mathbf{AX}\, g$ is not in s, then s does not satisfy $\mathbf{AX}\, g$. Hence, s may have successors that satisfy g and others that do not. The definition for the transition relation R_T is

$$R_T(s_1, s_2) = \bigwedge_{\mathbf{AX}\, g \in el(f)} s_1 \in sat(\mathbf{AX}\, g) \Rightarrow s' \in sat(g).$$

We must also add an acceptance condition to guarantee that *eventuality* properties are fulfilled. The acceptance condition should restrict the set of (fair) paths so that:

- For every (fair) path π, for every elementary formula $\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h]$ of f, and for every state s on π, if $s \in sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$ then there is a later state t on π such that $t \in sat(h)$.

This can be enforced by a Büchi acceptance condition because of the following observation. Let s be a state in $sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$ and let state t be a successor of s under R_T; then either $t \in sat(h)$ or $t \in sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$. Thus, if $s \in sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$, then all succeeding states must also be in $sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$ unless we reach a state in $sat(h)$. The only way s could be in $sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])$ but not satisfy $\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h]$ would be for there to exist some path from s where every state on the path was in the set $sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h]) \cap (S_T - sat(h))$. To keep such paths from being fair, we require that infinitely often we reach a state in the complement of this set, which is $(S_T - sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])) \cup sat(h)$. Thus, we obtain the following acceptance condition:

$$\mathcal{F} = \{\ ((S_T - sat(\mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h])) \cup sat(h)) \mid \mathbf{AX}\, \mathbf{A}[g\ \mathbf{U}\ h] \in el(f)\ \}.$$

The correctness of the tableau construction is guaranteed by the following lemma.

Lemma 14. *For all subformulas g of f, if $s \in sat(g)$, then $s \models g$.*

The proof of a similar lemma for a tableau that uses Streett acceptance condition is given in [29]. The main result of this lemma is that the tableau for f satisfies f. To see this, note that any initial state of $\mathcal{T}_f$ is in $sat(f)$ and therefore every initial state of $\mathcal{T}_f$ satisfies f.

An important property of this tableau construction is that any structure that satisfies f precedes $\mathcal{T}_f$ in the preorder $\preceq$. To show this we must define a simulation relation between the tableau and any structure that satisfies f. This is achieved by defining two states to be related if and only if they satisfy exactly the same set of subformulas of f. If we define H by

$$H = \{ (s', s) \mid s = \{ g \mid g \in el(f), s' \models g \} \}$$

then this property is guaranteed for the set of elementary formulas $el(f)$.

Lemma 15. *If $H(s', s)$, then for every subformula or elementary formula of f, $s' \models f$ implies $s \in sat(f)$.*

Lemma 16. *H is a simulation relation between $\mathcal{T}_f$ and the structure M'.*

The proof of these lemma also appears in [29].

Theorem 17. *For any structure M', $M' \models f$ if and only if $M' \preceq \mathcal{T}_f$.*

Proof. Suppose $M' \preceq \mathcal{T}_f$. By lemma 14 and the definition of the tableau, every initial state of $\mathcal{T}_f$ satisfies f, i.e., $\mathcal{T}_f \models f$. Then since $M' \preceq \mathcal{T}_f$, $M' \models f$.

If $M' \models f$, then by definition, every $s'_0 \in S'_0$ satisfies f. Let H be the relation defined above. By the definition of H, every such s'_0 is paired with a (unique) s_0. Lemma 15 implies that $s_0 \in sat(f)$, and by the definition of the tableau, $s_0 \in S_0$. By lemma 16, H is a simulation relation, so $M' \preceq \mathcal{T}_f$. □

The tableau construction can also be used to reason about formulas. We are typically interested in whether every model of a formula g is also a model of some other formula f. Let $g \models f$ denote this semantic relation.

Corollary 18. *$g \models f$ if and only if $\mathcal{T}_g \models f$.*

Proof. If $g \models f$, then every model of g, in particular $\mathcal{T}_g$, is also a model of f. Assume $\mathcal{T}_g \models f$, and let $M \models g$. By the previous theorem, $M \preceq \mathcal{T}_g$. By Theorem 9, $M \models f$. □

6.3 Justifying assume–guarantee proofs

The theory developed earlier in this section can be used to justify assume-guarantee proofs. We first consider a formal example given in an extension of Pnueli's notation [42]. The extension allows assumptions and specifications to be given either as formulas or directly as finite state models, whichever is more concise or convenient.

$$\frac{\begin{array}{c}\langle true\rangle M\langle A\rangle \\ \langle A\rangle M'\langle g\rangle \\ \langle g\rangle M\langle f\rangle\end{array}}{\langle true\rangle M \parallel M'\langle f\rangle}$$

In the proof, A, M, and M' are finite state models and g and f are ACTL formulas. In our framework, this corresponds to

$$\frac{\begin{array}{c}M \preceq A \\ A \parallel M' \models g \\ \mathcal{T}_g \parallel M \models f\end{array}}{M \parallel M' \models f}$$

The soundness of this rule is established by showing that the conclusion must be true if each of the three hypotheses is true.

1.	$M \preceq A$	Hypothesis
2.	$M \parallel M' \preceq A \parallel M'$	Theorem 12
3.	$A \parallel M' \models g$	Hypothesis
4.	$A \parallel M' \preceq \mathcal{T}_g$	Theorem 17
5.	$M \parallel M' \preceq \mathcal{T}_g$	Transitivity of $\preceq$
6.	$M \parallel M \parallel M' \preceq \mathcal{T}_g \parallel M$	Theorem 12
7.	$\mathcal{T}_g \parallel M \models f$	Hypothesis
8.	$M \parallel M \parallel M' \models f$	Transitivity of $\preceq$
9.	$M \preceq M \parallel M$	Theorem 13
10.	$M \parallel M' \preceq M \parallel M \parallel M'$	Theorem 12
11.	$M \parallel M' \models f$	Theorem 9

6.4 Verifying a CPU controller

A symbolic model checker based on the theory developed earlier in this section is described in [29]. It includes facilities for model checking, temporal reasoning (via the tableau construction), and checking if one structure simulates another. The model checker is used to verify a simple CPU controller. We give only a brief description of the CPU here; Clarke, Long and McMillan [21] give details. The CPU is a simple stack-based machine, i.e., part of the CPU's memory contains a stack from which instruction operands are popped and onto which results are pushed. There are two parts to the CPU controller. The first part is called the access unit and is responsible for all the CPU's memory references. The second part, called the execution unit, interprets the instructions and controls the arithmetic unit, shifter, etc. These two parts operate in parallel. The access unit and execution unit communicate via a small number of signals. Three of the signals, *push*, *pop* and *fetch*, are inputs of the access unit and indicate that the execution unit wants to push or pop something from the stack or to get the next instruction. For each of these signals there is a corresponding ready output from the access unit. The execution unit must wait for the appropriate ready signal before proceeding. One additional signal, *branch*, is asserted by the execution unit when it wants to jump to a new program location.

In order to increase performance, the access unit attempts to keep the value on the top of the stack in a special register called the TS register. The goal is to keep the execution

unit from having to wait for the memory. For example, when the TS register contains valid data, a pop operation can proceed immediately. In addition, when a value is pushed on the stack, it is moved into this register and copied to memory at some later point. The access unit also loads instructions into a queue when possible so that fetches do not require waiting for the memory. This queue is flushed whenever the CPU branches.

We divide the specifications that we check into three classes. The first class consists of simple safety properties; we omit the description of their verification here. The conditions in the second class are slightly more complex. These properties are safety properties which specify what sequences of operations are allowed. They depend on the access unit asserting the various ready signals at appropriate times and on the memory acknowledge signal being well-behaved. In order to verify the properties, we made a simple model of the memory which behaves as follows:

1. The memory waits for a read or write request.
2. An arbitrary but finite time later it produces an acknowledgment signal.
3. The acknowledgment signal is removed one cycle later.

By composing this model with the access unit, we were able to verify all of the properties except one. To verify the exception, an additional assumption $\mathbf{AG}(\neg push \vee \neg pop)$ was required. The model checker verified that the property was true under this assumption by building the tableau for the assumption, composing it with the access unit and memory model, and checking the property.

The final class consists of a single liveness property: $\mathbf{AG}\,\mathbf{AF}(fetch \wedge fetchrdy)$. This formula states that the CPU always fetches another instruction. One way to verify this property involves making a model of the execution unit. We describe an alternative way of doing the verification that uses a series of ACTL assumptions.

The idea will be to check the property for the execution unit. In order for the formula to be true, the access unit must eventually respond to push and pop requests and must fill the instruction queue when appropriate. We can only guarantee that the access unit meets these conditions if we know that the execution unit does not try to do two operations at once and that it will not remove a request before the corresponding operation is completed. We begin with these properties.

$$\mathbf{AG}(\neg(fetch \wedge push) \wedge \neg(fetch \wedge pop) \wedge \cdots \wedge \neg(pop \wedge branch)) \tag{1}$$

$$\mathbf{AG}(push \rightarrow \mathbf{A}[pushed\ \mathbf{V}\ push]) \tag{2}$$

$$\mathbf{AG}(pop \rightarrow \mathbf{A}[popped\ \mathbf{V}\ pop]) \tag{3}$$

The first of these specifies that every pair of operations the execution unit can perform are mutually exclusive. The other two formulas state that if the execution unit makes a push or pop request, then it does not deassert the request until the operation completes. The model checker verified that these properties hold in the execution unit alone, and (using the tableau construction) that the first property implies the assumption $\mathbf{AG}(\neg push \vee \neg pop)$ used above. Now using formulas 1 and 2 as assumptions, we checked that the system composed of the access unit and the memory model satisfied the formula

$$\mathbf{AG}(push \rightarrow \mathbf{A}[push\ \mathbf{U}\ pushed]). \tag{4}$$

This specification states that every push operation will be completed. Similarly, using formulas 1 and 3 as assumptions, we verified

$$\mathbf{AG}(pop \rightarrow \mathbf{A}[pop\ \mathbf{U}\ popped]). \tag{5}$$

The system composed of the access unit and the memory model also satisfies the formula $\mathbf{AG}\,\mathbf{AF}(fetchrdy \vee branch)$ (at any point, either the access unit will eventually fill the instruction queue or a branch will occur). Finally, using this formula and formulas 4 and 5 as assumptions, the model checker verified that the execution unit satisfies $\mathbf{AG}\,\mathbf{AF}(fetch \wedge fetchrdy)$. To complete the verification and conclude that the entire specification was true of a system, we would have to check that the actual memory was simulated by the model we used.

7 Abstraction

In this section, we discuss techniques for verifying programs with data-dependent behavior. In order to describe such programs, it is convenient to start with a slightly different model of computation in which the data is represented explicitly. Let the variables of the program be $x_1, x_2, \ldots, x_n$, and let each variable x_i have a corresponding domain of values D_{x_i}. The set of possible program states S is simply the cross product $D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$. The *state transition graph* of the program is a triple $T = (S, S_0, R)$, where $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a transition relation. We could add a fairness constraint as well, but for simplicity we will ignore fairness throughout this section.

In order to use the theory and specification techniques that we have developed earlier, we need to convert state transition graphs like the above to Kripke structures. To do this, we need to decide what information about the program variables should be made visible in the atomic propositions of the structure. We will only be able to specify and verify properties concerning the information visible in this way. For example, we could create an atomic proposition for each program variable x_i and value d_i in D_{x_i}. Denote this atomic proposition by '$x_i = d_i$'. Then a state where x_1 has the value 7 is labeled by the proposition '$x_1 = 7$'.

Unfortunately, due to the large (or perhaps even infinite) state space of interesting programs, we cannot directly apply our automated verification techniques. Furthermore, if the exact values of all the program variables are visible, the state space reduction techniques described earlier is ineffective. Thus, we must hide some of this information. For this reason, we map the possible values for each program variable into a small number of abstract values. For example, suppose x is a variable and the domain D_x is all integers. Assume that we are interested in expressing a property involving the sign of x. We create a domain A_x of abstract values for x, with $A_x = \{a_0, a_+, a_-\}$, and define a mapping h_x from D_x to A_x as follows:

$$h_x(d) = \begin{cases} a_0, & \text{if } d = 0, \\ a_+, & \text{if } d > 0, \text{ and} \\ a_-, & \text{if } d < 0. \end{cases}$$

Now we use just three atomic propositions to express the abstract value of x. We will denote these propositions by '$\hat{x} = a_0$', '$\hat{x} = a_+$', and '$\hat{x} = a_-$', where $\hat{x}$ indicates that we

are referring to abstract rather than actual values. Note that we can no longer express properties about the exact actual value of x using these atomic propositions. In many cases though, by judicious choice of the abstraction mapping, knowing just the abstract value is sufficient. If we apply this abstraction process to some of the program variables, the Kripke structure obtained has a smaller number of atomic propositions. Let M be this structure. Now state space reduction techniques may be applicable to reduce the complexity of verification.

The particular technique that we will use is based on simulation relations. The idea will be to merge together all states that have the same labeling of (abstract-level) atomic propositions. Thus, in the reduced structure, every state will have a unique labeling. Consequently, the labeling can be used to identify the state. In the above example, we would collapse all states labeled with '$\hat{x} = a_+$' into one state, i.e., all states where $x > 0$ are merged into one. We denote this collapsed state by the label '$\hat{x} = a_+$' as well. When we do the collapsing, we must ensure that the reduced structure simulates the original one. So, if there is a transition in M between states corresponding to $x = 0$ and $x = 5$, in our reduced system we must add a transition between the states labeled with '$\hat{x} = a_0$' and '$\hat{x} = a_+$'. Similarly, if the state where $x = -7$ is an initial state, we must make the state labeled with '$\hat{x} = a_-$' an initial state. Formally, we collapse M to the structure M_r ("r" for "reduced state space") defined as follows:

1. $S_r = \{ L(s) \mid s \in S \}$. That is, S_r is the set of all labelings of states of M.
2. $s_r \in S_0^r$ if and only if there exists s such that $s_r = L(s)$ and $s \in S_0$.
3. $AP_r = AP$.
4. Each s_r is just a set of atomic propositions, so we take $L_r(s_r) = s_r$.
5. $R_r(s_r, t_r)$ if and only if there exist s and t such that $s_r = L(s)$, $t_r = L(t)$, and $R(s, t)$.

It is now easy to see that the reduced structure M_r simulates the original structure M, because we can use $H = \{ (s, s_r) \mid s_r = L(s) \}$ as a simulation relation. Thus, whatever ACTL* properties we can prove about M_r will also hold in M. We think of M_r as an abstract version of the state transition graph of the program. The states of M_r correspond to mappings from the program variables to abstract values. Each abstract state represents a set of concrete states that are merged together during the collapsing process.

Figure 5 illustrates the abstraction process for a simple traffic light controller. The original program has one variable *color* that can take on the values *red*, *yellow*, and *green*. The structure M is obtained by mapping *red* and *yellow* to *stop* and *green* to *go*. The reduced structure M_r results from merging together those states of M with the same labeling of atomic propositions .

We can use M_r to deduce properties of the program since M_r simulates M. The difficulty is that building M_r requires constructing M, and this is often impractical. In the cases where we cannot build M, we use the fact that we do have an implicit representation of M, namely, the program text. We will avoid dealing with M by directly compiling the program to an abstract structure. Unfortunately, we will usually not be able to produce exactly M_r, but we will produce an approximation M_a that simulates M_r ("a" for "approximation"). The goal is to have M_a sufficiently close to M_r so that we can still verify interesting properties of the program.

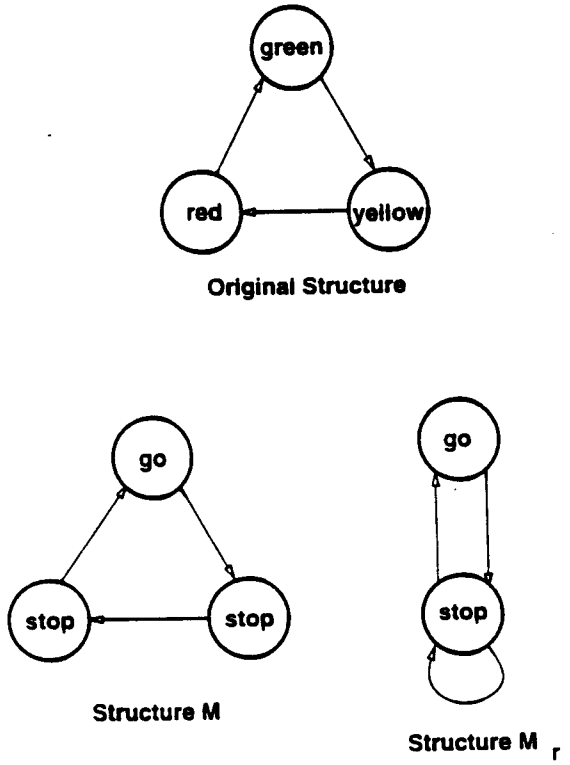

Fig. 5. Traffic light controller example

7.1 Compilation

In order to describe how M_a is produced, we must consider the process of compiling a program into a state transition graph. To keep the discussion relatively independent of the particular programming language used, we will argue that a program can be transformed into *relational expressions* $\mathcal{S}_0$ and $\mathcal{R}$ that can be evaluated to obtain the initial states S_0 and the transition relation R of T. These relational expressions are simply first-order formulas that will be built up from a set of *primitive relations* for the basic operators and constants in the language. Later, we will use these formulas to obtain an abstract state transition graph which will give us M_a. 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.

In the remainder of this subsection, we consider how $\mathcal{S}_0$ and $\mathcal{R}$ can be derived. Since this is not the main concern of the section, we will just consider an example program (Figure 6). 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 ($lsb(b)$) 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). 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.

```
0: p ← 0
1: while b ≠ 0
       p ← p ⊕ lsb(b)
       b ← b ≫ 1
   endwhile
2: end
```

Fig. 6. 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. In order 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:

$$PC = 0 \wedge p' = 0 \wedge b' = b \wedge PC' = 1.$$

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:

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

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 (note that the first two lines are just the point to point relations derived above):

$$(PC = 0 \wedge p' = 0 \wedge b' = b \wedge PC' = 1) \vee$$
$$(PC = 1 \wedge b = 0 \wedge p' = p \wedge b' = b \wedge PC' = 2) \vee$$

$$(PC = 1 \wedge b \neq 0 \wedge p' = p \oplus lsb(b) \wedge b' = b \gg 1 \wedge PC' = 1) \vee$$
$$(PC = 2 \wedge p' = p \wedge b' = b \wedge PC' = 2).$$

Now 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. In order 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 $p' = p \oplus lsb(b)$. 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 $p' = p \oplus lsb(b)$ 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 $\mathcal{S}_0$ describing the initial conditions on p, b, and PC could be derived in a similar way.

In general, the derivation of $\mathcal{S}_0$ and $\mathcal{R}$ are 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 section; 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.

7.2 Computing approximations

Throughout this subsection, 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 $x_1, x_2, \ldots$, range over the same domain D. We use a set $\widehat{x_1}, \widehat{x_2}, \ldots$, of variables ranging over the abstract domain A, 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 A.

We use $\mathcal{S}_0$ and $\mathcal{R}$ of the previous section to define the state transition graph $T = (S, S_0, R)$ with state set $S = D \times \cdots \times D$. S_0 is the set of valuations that satisfy the formula $\mathcal{S}_0$. Similarly, the transition relation R is derived from the formula $\mathcal{R}$. Recall that we use T to produce the Kripke structure M. In particular, we take S, S_0, and R for the Kripke structure to be the same as in T. The labeling function L is defined as follows. Let $s = (d_1, \ldots, d_n)$, i.e., in state s, x_i has value d_i. Define $a_i = h(d_i)$. We introduce an atomic proposition '$\widehat{x_i} = a_i$' to denote that x_i has the abstract value a_i. Now $L(s) = \{\text{'}\widehat{x_1} = a_1\text{'}, \ldots, \text{'}\widehat{x_n} = a_n\text{'}\}$.

To produce M_a, we will require a state transition graph over the abstracted state set $A \times \cdots \times A$. First, we note that we can obtain an abstract state transition graph T_r corresponding to M_r by evaluating the formulas

$$\widehat{S_0} = \exists x_1 \ldots \exists x_n \, (h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_n) = \widehat{x_n} \wedge S_0(x_1, \ldots, x_n)).$$

and

$$\widehat{\mathcal{R}} = \exists x_1 \ldots \exists x_n \exists x'_1 \ldots \exists x'_n \, (h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_n) = \widehat{x_n} \wedge h(x'_1) = \widehat{x_1}' \wedge \cdots \wedge h(x'_n) = \widehat{x_n}' \wedge \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n)).$$

For conciseness, we will denote this existential abstraction operation by $[\cdot]$. If ϕ depends on the free variables $x_1, \ldots, x_m$, 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 $x_1, \ldots, x_m$. So the abstract state transition graph corresponding to M_r is given by the formulas $\widehat{S_0} = [S_0]$ and $\widehat{\mathcal{R}} = [\mathcal{R}]$.

Ideally, we would like to extract S_0^r and R_r from $[S_0]$ and $[\mathcal{R}]$ directly. However, this is often computationally expensive. Thus, we will now define a transformation $\mathcal{C}$ on formulas ϕ. The idea of $\mathcal{C}$ 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. $\mathcal{C}(P(x_1, \ldots, x_m)) = [P](\widehat{x_1}, \ldots, \widehat{x_m})$ and $\mathcal{C}(\neg P(x_1, \ldots, x_m)) = [\neg P](\widehat{x_1}, \ldots, \widehat{x_m})$ if P is a primitive relation.
2. $\mathcal{C}(\phi_1 \wedge \phi_2) = \mathcal{C}(\phi_1) \wedge \mathcal{C}(\phi_2)$.
3. $\mathcal{C}(\phi_1 \vee \phi_2) = \mathcal{C}(\phi_1) \vee \mathcal{C}(\phi_2)$.
4. $\mathcal{C}(\exists x \, \phi) = \exists \hat{x} \, \mathcal{C}(\phi)$.
5. $\mathcal{C}(\forall x \, \phi) = \forall \hat{x} \, \mathcal{C}(\phi)$.

In other words, $\mathcal{C}$ pushes the existential quantifications inwards so that the abstraction operation $[\cdot]$ is only applied at the innermost level. Since these inner abstractions are relatively simple, they can be evaluated easily. Thus, while we may not be able to evaluate $[S_0]$ and $[\mathcal{R}]$, we generally can evaluate $\mathcal{C}(S_0)$ and $\mathcal{C}(\mathcal{R})$. This will yield the state transition graph $T_a = (S_a, S_0^a, R_a)$. We define M_a using T_a. As when we defined M from T, we take the state set, initial states, and transition relation directly from T_a. L_a is defined as follows. Let $s_a = (a_1, \ldots, a_n) \in S_a$. Then $L_a(s_a) = \{\text{`}\widehat{x_1} = a_1\text{'}, \ldots, \text{`}\widehat{x_n} = a_n\text{'}\}$. Note that $s = (d_1, \ldots, d_n) \in S$ and s_a will be identically labeled if for all i, $h(d_i) = a_i$.

The price that we pay for simplifying the evaluation is that we may add extra initial states and transitions to the corresponding state transition graph. This is why M_a will only be an approximation to M_r. On the other hand, in order to know that M_a simulates M_r, we must show that applying $\mathcal{C}$ cannot cause us to lose any initial states or transitions. This is a consequence of the following theorem.

Theorem 19. *$[\phi]$ implies $\mathcal{C}(\phi)$. In particular, $[S_0] \rightarrow \mathcal{C}(S_0)$ and $[\mathcal{R}] \rightarrow \mathcal{C}(\mathcal{R})$.*

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 $[\phi] = \mathcal{C}(\phi)$ and the theorem holds.
2. Let $\phi(x_1, \ldots, x_m) = \phi_1 \wedge \phi_2$. Then, $[\phi_1 \wedge \phi_2]$ is identical to the formula

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

   This formula implies

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

   which is exactly $[\phi_1] \wedge [\phi_2]$. Now $\mathcal{C}(\phi_1 \wedge \phi_2) = \mathcal{C}(\phi_1) \wedge \mathcal{C}(\phi_2)$, and by the induction hypothesis we have $[\phi_1]$ implies $\mathcal{C}(\phi_1)$ and $[\phi_2]$ implies $\mathcal{C}(\phi_2)$. Hence $[\phi_1] \wedge [\phi_2]$ implies $\mathcal{C}(\phi_1) \wedge \mathcal{C}(\phi_2)$, and so $[\phi_1 \wedge \phi_2]$ implies $\mathcal{C}(\phi_1 \wedge \phi_2)$.
3. The case where $\phi = \phi_1 \vee \phi_2$ is similar to the previous case.
4. Let $\phi(x_1, \ldots, x_m) = \forall x \phi_1$. Then $[\forall x \phi_1]$ is

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

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

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

   This implies

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

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

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

   This is exactly $\forall \hat{x} [\phi_1]$. Now by the induction hypothesis, $[\phi_1]$ implies $\mathcal{C}(\phi_1)$, and so $\forall \hat{x} [\phi_1]$ implies $\forall \hat{x} \mathcal{C}(\phi_1)$. This latter formula is equal to $\mathcal{C}(\forall x \phi_1)$.
5. The case where $\phi = \exists x \phi_1$ is similar to the previous case. □

The above idea of "pushing the abstractions inwards" is essentially the same one that is used in abstract interpretation [23, 24, 38, 39]. By defining a suitable abstract domain of computation and then interpreting a program relative to this domain, it is possible to extract information about the program. The goal is usually to obtain data that will be used at compile time to optimize the program. Abstract interpretations have been defined for applications such as: strictness and reference count analysis for functional programs; finding linear relationships between variables; and computing live ranges for variables. The interpretation is done using abstract versions of the language operators. These abstract operators correspond to the abstract versions of the primitive relations

above. The relationships between abstract interpretation and the use of abstraction in model checking are discussed in more detail elsewhere [4, 26].

Finally, we show that M_a simulates M. This is the basis for using abstraction to verify properties of the program.

Theorem 20. $M \preceq M_a$.

Proof. We show this by giving a simulation relation between M and M_a. Let $s = (d_1, \ldots, d_n)$ and $s_a = (a_1, \ldots, a_n)$. Define $H(s, s_a)$ if and only if for all i, $h(d_i) = a_i$.

Assume $H(s, s_a)$, with $s = (d_1, \ldots, d_n)$. Then $s_a = (h(d_1), \ldots, h(d_n))$. Note first that the two states have the same labeling. Recall that the label of s is the set of propositions '$\widehat{x_i} = a_i$', where the value of x_i is mapped to a_i by h. Assume then that '$\widehat{x_i} = a_i$' labels s. This is true if and only if $a_i = h(d_i)$. Now s_a will be labeled by '$\widehat{x_i} = a_i$' if and only if the ith component of s_a is a_i. But the ith component of s_a is $h(d_i)$, which is equal to a_i.

Assume $R(s, t)$, where $t = (e_1, \ldots, e_n)$. Define $t_a = (h(e_1), \ldots, h(e_n))$. We must show that $R_a(s_a, t_a)$. By the definition of R, we know that s and t correspond to valuations satisfying $\mathcal{R}$. Now we show that $[\mathcal{R}](s_a, t_a)$. By definition of $[\cdot]$, we have $[\mathcal{R}](s_a, t_a)$ if and only if

$$\exists x_1 \ldots \exists x_n \exists x'_1 \ldots \exists x'_n \left(\bigwedge_{i=1}^{n} (h(x_i) = h(d_i) \wedge h(x'_i) = h(e_i)) \wedge \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \right).$$

Since $\mathcal{R}(s, t)$, we can see that the above formula holds by taking the d_i as witnesses for the x_i and the e_i as witnesses for the x'_i. Now that we know $[\mathcal{R}](s_a, t_a)$, the previous theorem implies that $\mathcal{C}(\mathcal{R})(s_a, t_a)$. But $\mathcal{C}(\mathcal{R})$ defines R_a, so $R_a(s_a, t_a)$. Thus, H is a simulation relation between M and M_a.

Using a similar argument, we see that if $s \in S_0$, then $s_a \in S_0^a$. Thus every initial state of M has a corresponding initial state of M_a, and so $M \preceq M_a$. □

7.3 Exact approximations

In the previous section, we showed that $M \preceq M_a$; thus, every ACTL* formula satisfied by M_a also holds in M. In this subsection, we consider some additional conditions that allow us to show M is bisimulation equivalent to M_a. These conditions permit us to verify any CTL* formula using M_a and conclude that M also satisfies the formula. When these conditions are satisfied, we will call M_a an exact approximation of M.

To begin, we note that each abstraction mapping h_x for the program variable x induces an equivalence relation $\sim_x$ defined as follows. Let d_1 and d_2 be in D_x. Then $d_1 \sim_x d_2$ if and only if $h_x(d_1) = h_x(d_2)$. The key condition for exact approximations is that these equivalence relations are *congruences* for the primitive relations corresponding to the basic operations used in the program. Recall the definition of congruence: Let $P(x_1, \ldots, x_m)$ be a relation with x_i ranging over D_{x_i}. The equivalence relations $\sim_{x_i}$ are a congruence with respect to P if and only if

$$\forall d_1 \ldots \forall d_m \forall e_1 \ldots \forall e_m \left(\bigwedge_{i=1}^{m} d_i \sim_{x_i} e_i \rightarrow (P(d_1, \ldots, d_m) \Leftrightarrow P(e_1, \ldots, e_m)) \right).$$

The proof of exactness when the $\sim_{x_i}$ are congruences with respect to the primitive relations is based on the following theorem which is the analog of Theorem 19. This theorem allows us to show that M_{r} and M_{a} are bisimulation equivalent.

Theorem 21. *If the $\sim_{x_i}$ are congruences with respect to the primitive relations and o is a relational expression defined over these relations, then $[\phi] \Leftrightarrow \mathcal{C}(\phi)$. In particular $[S_0]$ and $[\mathcal{R}]$ are equivalent to $\mathcal{C}(S_0)$ and $\mathcal{C}(\mathcal{R})$, respectively.*

This is used to prove bisimulation equivalence between M and M_{a}.

Theorem 22. *If $\sim_{x_i}$ are congruences with respect to the primitive relations, then $M \equiv M_{\mathrm{a}}$.*

For the proofs of these theorems, see Long [20]. The intuition is that we can lift the congruence on primitive relations to a congruence on any relational expression built from these primitives. Now let s_1 and s_2 be states of M that have the same labeling. Then they are related by the $\sim_{x_i}$. By the previous observation, if s_1 and t_1 satisfy $\mathcal{R}$, then s_2 and t_2 must also satisfy $\mathcal{R}$ for every t_2 that is related by $\sim_{x_i}$ to t_1. Thus s_1 and s_2 are bisimilar, so all states which are collapsed to one state in forming M_{r} are bisimilar to each other, and to the state in M_{r}. This in turn implies that M and M_{r} are bisimulation equivalent. The previous theorem is then used to prove that $M \equiv M_{\mathrm{a}}$.

7.4 A simple language

We now describe a verification system based on the ideas in the previous subsections. The system consists of a compiler for a finite state language, plus a BDD-based model checker. In this section, we briefly describe the language, which is designed for specifying reactive programs. The main features of this language are:

1. It is procedural and 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 [37] for implementation in hardware. A Moore machine is a standard model for synchronous circuits.

and for verification there is a standard transformation of Moore machines into Kripke structures [2]. When abstraction is not used, our compiler produces this Kripke structure directly. Since the structure may have a large number of states, it is important not to generate an explicit-state representation. Instead, the compiler directly produces a description of the structure in the form of a BDD. This is then used as the input to the model checking program.

To use abstraction, the user specifies abstractions for some of the variables at the time of compilation. By using the techniques described in the previous subsections, the compiler directly generates an abstract structure. There are a number of abstractions built into the compiler, some of which are described in the following section. 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 7 is a small example program, a settable countdown timer, written in the language. The timer has two input variables, *set* and *start*, which are one and eight bits wide respectively. There are also two output variables: *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.

```
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. 7. An example program

7.5 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. For example, if x is a variable that is abstracted by:

$$h(d) = \begin{cases} a_{\mathbf{even}}, & \text{if } d \text{ is even;} \\ a_{\mathbf{odd}}, & \text{if } d \text{ is odd,} \end{cases}$$

then we will generally write something like $even(x)$ in a formula rather than "$\hat{x} = a_{\mathbf{even}}$".

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 23 Chinese remainder theorem. *Let $m_1, m_2, \ldots, m_n$ be positive integers which are pairwise relatively prime. Define $m = m_1 m_2 \ldots m_n$, and let $b, i_1, i_2, \ldots, i_n$ 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 $m_1, m_2, \ldots, m_n$. 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 8). 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

```
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. 8. A 16 bit multiplier

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]

[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

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

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j) \rightarrow \mathbf{A}[\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge (\mathit{overflow} \vee (\mathit{output} \bmod m = ij \bmod m))])$$

Here, i and j range from 0 through $m-1$, 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 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.

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 8. 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.

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \leq 16) \rightarrow \mathbf{A}[\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge \neg \mathit{overflow}])$$

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \geq 18) \rightarrow \mathbf{A}[\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge \mathit{overflow}])$$

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.

---

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 7.5).

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

$$h(i) = (H_1(i), H_2(i))$$

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.

As an example of using these types of abstractions, consider the program shown in Figure 9. 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. $\sharp b \oplus parity$ is invariant from that point onwards.

We can express the above with the following formula.

$$\neg\sharp in \wedge \mathbf{AX}(\neg\sharp b \wedge \mathbf{AG}\,\neg(\sharp b \oplus parity)) \vee \sharp in \wedge \mathbf{AX}(\sharp b \wedge \mathbf{AG}(\sharp b \oplus parity))$$

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.

```
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. 9. A parity computation program

7.6 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 10. We wish to show that the next state value of b is always equal to the current state value

```
input a : 8
output b : 8 ← 0

loop
    b ← a
    wait
endloop
```

Fig. 10. A simple program

of a. We can express this property for a fixed value, say 42, using the formula:

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\, b = 42).$$

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:

$$\mathbf{AG}(\widehat{a} = 0 \rightarrow \mathbf{AX}\, \widehat{b} = 0).$$

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

$$\mathbf{AG}(\widehat{a} = 0 \rightarrow \mathbf{AX}\, \widehat{b} = 0)$$

is essentially saying

$$\mathbf{AG}(a = c \rightarrow \mathbf{AX}\, b = c).$$

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

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\, b = 42).$$

In general, we have found that sharing in the BDDs makes it possible to perform the abstraction, compilation, and model checking efficiently. We call abstractions such as h_c "symbolic abstractions".

We used symbolic abstractions to verify a simple pipeline circuit. This circuit is shown in Figure 11 and is described in detail elsewhere [12, 14]. 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. 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

$$\begin{aligned}\mathbf{AG}((&srcaddr1 = ra) \wedge (srcaddr2 = rb) \wedge (destaddr = rc) \wedge \neg stall \\ &\rightarrow \mathbf{AX}\,\mathbf{AX}((regra = ca) \wedge (regrb = cb) \rightarrow \mathbf{AX}(regrc = ca + cb))).\end{aligned}$$

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 nearly 10^{1300}

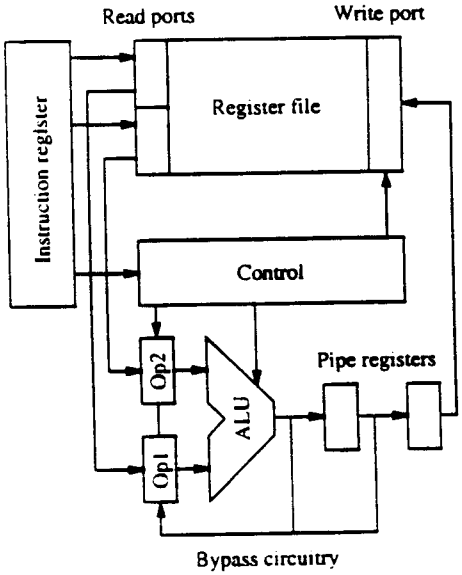

Fig. 11. Pipeline circuit block diagram

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.* [12] 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 ACTL*(see Section 5). By combining symbolic compositions with symbolic abstractions, we were able to verify the system 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 [34].

8 Directions for Future Research

There are several directions for future work. Certainly the most important question is whether symbolic model checking algorithms are always superior to model checking algorithms that use an explicit representation for the state-transition graph. It appears that the symbolic algorithms are rarely worse than the explicit state algorithms and in many cases are significantly better. However, the definitive answer to this question can only be determined by additional experimentation.

Another obvious question is whether it is possible to apply abstraction and compositional reasoning techniques to branching-time logics with *existential path quantifiers*. The restriction to ACTL can produce misleading results. For example, an ACTL formula may be vacuously true if there are no fair paths. This is a serious problem. If the theory that has been developed could be extended to full CTL, this problem might be avoided.

There are several technical problems that should be relatively easy to solve. The compositional reasoning and abstraction techniques that we have developed apply only to synchronous systems. Although synchronous systems are probably more common in hardware design, asynchronous systems are extremely important for reasoning about distributed systems. We see no inherent reason why the techniques described in this paper cannot be extended to handle both types of parallel composition. Another important problem is to develop efficient algorithms for checking the simulation preorder between arbitrary structures which may involve fairness constraints. This will make the theory developed in Sections 6 and 7 much more widely applicable.

Perhaps the most exciting problem is to extend the theory that has developed to *infinite state systems*, so that software can be handled as well [31, 34]. Data values might be represented as terms in the Herbrand universe determined by the program. Given an equivalence relation of finite index on these terms, we should be able to derive abstractions for the primitive relations and use these to produce an abstract version of the program. However, this will require considerably more thought.

Acknowledgements

We are also grateful to Jurgen Dingel, Rajit Manohar, and Will Marrero for their careful reading of this paper.

References

1. A.Aziz, V. Singhal, F. Balarin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Equivalences for fair kripke structures. In *Proc. 21st Int. Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, Springer-Verlag, July 1994.
2. S. A.Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.
3. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
4. 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*, July 1992.
5. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In DAC90 [25].
6. M. C. Browne and E. M. Clarke. Sml: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France*. IFIP, September 1986.
7. M. C. Browne, E. M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proceedings of the 1985 International Conference on Computer Design*, Port Chester, New York, October 1985. IEEE.

8. M. C. Browne, E. M. Clarke, and D. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design*. Elsevier Science Publishers (North Holland), 1986.
9. 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, 1986.
10. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.
11. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
12. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1991.
13. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991. Winner of the Sidney Michaelson Best Paper Award.
14. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In DAC90 [25].
15. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1990.
16. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.
17. E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1990.
18. 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.
19. 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.
20. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. January 1992.
21. E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In J. A. Darringer and F. J. Rammig, editors, *Proceedings of the Ninth International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, June 1989.
22. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
23. 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.
24. 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.

25. *Proceedings of the 27th ACM/IEEE Design Automation Conference.* IEEE Computer Society Press, June 1990.
26. D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of ctl. Submitted for publication.
27. D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.
28. E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.
29. O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, May 1994.
30. Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.-Feb. 1990.
31. D. Jackson. Abstract model checking of infinite specifications. In *Proceedings of Formal Methods Europe*, Barcelona, Oct. 1994.
32. R. P. Kurshan. Analysis of discrete event coordination. In 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.
33. 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.
34. D. L. Long. *Model Checking, Abstraction, and Compositional Reasoning.* PhD thesis, Carnegie Mellon University, 1993.
35. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.
36. B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.
37. E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton University Press, 1956.
38. A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.
39. F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.
40. R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing*, 16(6), Dec 1987.
41. C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. Clarke, editors, *Proc. CAV Workshop (also DIMACS Tech. Report 90-31)*, Rutgers University, NJ, June 1990.
42. A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F, Computer and system sciences*. Springer-Verlag, 1984.
43. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
44. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.
45. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.