Symbolic Model Checking of Infinite State Programs Using Presburger Arithmetic *

Tevfik Bultan, Richard Gerber and William Pugh
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
(301) 405-2710
{bultan,rich,pugh}@cs.umd.edu

ABSTRACT
Model checking is a powerful technique for analyzing large, finite-state systems. In an infinite transition system, however, many basic properties are undecidable. In this paper we present a new symbolic model checker which conservatively evaluates safety and liveness properties on infinite-state programs. We use *Presburger formulas* to symbolically encode a program's transition system, as well as its model-checking computations. All fixpoint calculations are executed symbolically, and their convergence is guaranteed by using approximation techniques. We demonstrate the promise of this technology on some well-known infinite-state concurrency problems.

Keywords
Static analysis, symbolic model checking, transition systems, Presburger arithmetic.

INTRODUCTION

In recent years CTL model checking [5] has emerged as a successful method for verifying large finite-state systems [4, 18]. Two main reasons behind this success are: (1) many of the properties one wants to check are representable in CTL; and (2) there are efficient procedures to check them. But when transition systems are not restricted to be finite, CTL model checking becomes undecidable.

We have developed a new symbolic model checker to attack this problem. Our technique is based on the following ideas:

- It symbolically encodes transition relations and program states using Presburger formulas, and uses this representation for computing truth sets of temporal logic formulas.
- It uses approximation techniques in its analysis of infinite-state programs, which guarantee convergence (by allowing false negatives).

*This work was supported in part by ONR grant N00014-94-10228, NSF YI CCR-9357850, NSF CCR-9157384 and a Packard Fellowship.

Of course, in any computer system variables are eventually mapped to finite representations. Thus it might be argued that integers can be given a finite range, and programs can then be analyzed as finite-state machines – for example, using BDD's [4, 14]. For two reasons, however, this may not always be the best way to proceed. First, mapping integer variables and operations to their binary implementations may lead to highly inefficient static analysis. But perhaps more importantly, one may wish to analyze an algorithm as an abstraction, and prove its correctness in a general sense, for any implementation of integers.

In this paper we demonstrate our model checker's effectiveness on some "classical" infinite-state programs, taken from a standard concurrency textbook [2]. While relatively small, they possess some interesting subtleties, especially in the tricky way their infinite-state variables influence control flow.

Other methods have been proposed to deal with infinite-state programs like these, and we note some of them here. One is to come up with a finite abstraction of the program, and then check the property on the abstraction [6]. Of course this requires the user to find the abstraction, and hence is not completely automatable. Another approach is to use a technique like symbolic execution [10, 11], which symbolically generates a program's execution paths. In practice, this method may end up generating an infinite number of nodes, and thus never terminate. This limitation can be overcome by having the user specify assertions about a process's behavior, which can be verified locally. Then the local proofs can be checked for cooperation [10]. Although the method has the benefit of incrementally proving correctness (as opposed to generating all possible interleavings), it relies on users to come up with the right assertions.

Our work has some strong historical antecedents. For example, Cooper developed a technique which encodes transition relations as sets of Presburger formulas, and then converts queries about a program's properties to validity checks in the Presburger arithmetic [7]. Due to the complexity of general Presburger solvers, however, proving correctness as single Presburger decision problem is not a method that can scale very well. We have found it more beneficial to use model checking as our primary technology, and use a Presburger solver for some subservient set-theoretic computations.

Our work was also influenced by known techniques from abstract interpretation [8]; specifically, we use some approximation methods first developed for that domain. Most properties of programs can be formulated as least fixpoints over sets of the program's states, and conservative abstract interpretation provides a way of approximating these fixpoints.

Our encoding of program states is similar to that used by Alur *et al.* in verifying Hybrid systems [1]. The fundamental difference is that we encode sets of integers (as opposed to the real numbers used in hybrid systems), and we can thus use Presburger formulas as our symbolic representation. This enables us to state properties such as, x *is even*, $x \leq y + z + 10$, etc. In hybrid systems, variables change linearly with respect to time, and range over the reals.

The remainder of the paper is organized as follows. In the following section we present the syntax, semantics, and Presburger encodings for concurrent programs and their properties. Then we describe our symbolic model checker, and show how it exploits the Presburger representation. After formally defining the term *conservative approximation*, we discuss the specific approximation techniques we use for computing upper and lower bounds of fixpoints. Finally, we conclude with some discussion on our results.

PROGRAMS AND PROPERTIES

In this section we define our models for representing programs and properties. We use the event-action language from [17] as our syntax for concurrent programs, with a semantics defined in terms of infinite transition systems. For representing temporal properties, we use a variant of CTL, in which the ground propositions are encoded as Presburger formulas.

Representing Programs

A concurrent program is represented by a (1) finite set of data and control variables, and (2) a finite set of events, where each event is considered atomic. The current state of a program is determined by the values of its data and control variables, in which the domain of each variable is countable. Each event is represented with an enabling condition and an action, where the enabling condition constrains the states in which the event can occur, and the action defines a transformation on the states.

Consider the concurrent program shown in Figure 1, which implements the *bakery algorithm* [2] to achieve mutual exclusion between two processes. Figure 2 shows the structure of the program, where arcs represent events and nodes represent control points. Note that actions of events are represented as relations, with v and v' denoting the current and next values of a variable v. If v is not mentioned in the action of an event, then we assume that $v' = v$. In the above program, control points for each process are denoted T, W, C, which stand for *thinking, waiting* or *in critical section*, respectively.

Data Variables: a, b: integer
Control Variables: $pc_1 : \{T_1, W_1, C_1\}$, $pc_2 : \{T_2, W_2, C_2\}$
Initial Condition: $a = b = 0$
Events:

e_{T_1}	**enabled:**	$pc_1 = T_1$
	action:	$pc_1' = W_1 \wedge a' = b + 1$
e_{W_1}	**enabled:**	$pc_1 = W_1 \wedge (a < b \vee b = 0)$
	action:	$pc_1' = C_1$
e_{C_1}	**enabled:**	$pc_1 = C_1$
	action:	$pc_1' = T_1 \wedge a' = 0$
e_{T_2}	**enabled:**	$pc_2 = T_2$
	action:	$pc_2' = W_2 \wedge b' = a + 1$
e_{W_2}	**enabled:**	$pc_2 = W_2 \wedge (b < a \vee a = 0)$
	action:	$pc_2' = C_2$
e_{C_2}	**enabled:**	$pc_2 = C_2$
	action:	$pc_2' = T_2 \wedge b' = 0$

Figure 1: The Bakery Algorithm

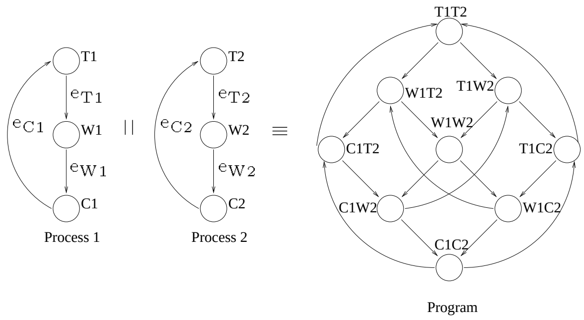

Figure 2: Structure of the bakery algorithm.

When a process wants to enter the critical section it first gets a ticket, which will be higher than that of all other processes currently in the critical section or waiting for entry. In the above program, variables a and b hold the ticket values for processes 1 and 2, respectively. A process gets its ticket by simply adding one to the highest outstanding ticket number.

There are two properties we wish to prove about the program. The first is mutual exclusion, i.e., *two processes are never in the critical section at the same time.*

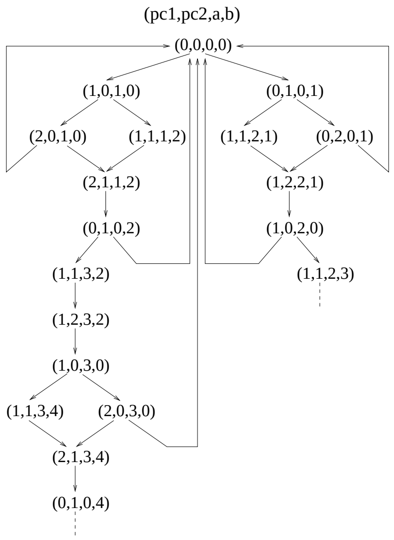

Figure 3: Part of the infinite transition system of the bakery algorithm.

The second is starvation-freedom, i.e., *if a process starts waiting for entry to the critical section, it eventually gets in.* The first property is a safety property, while the second is a progress property.

Note that variables a and b can increase without bound. For example, consider the scenario where process 1 gets a ticket; then process 2 gets a ticket before process 1 leaves the critical section. Then 1 gets another ticket before 2 leaves, etc. In other words, this is not a finite-state program.

Given a program in the above language, we model it as an infinite transition system $M = (S, I, X, L)$, where S is the domain of states, I is the set of initial states, $X \subseteq S \times S$ is the transition relation, and $L : S \times SF \rightarrow \{\mathsf{True}, \mathsf{False}\}$ is the valuation function for state formulas over the program's variables. These formulas form the building blocks for our temporal logic.

Figure 3 shows a part of the infinite transition system for the bakery algorithm above. Note that we use an interleaving model, where each transition represents execution of a single event, i.e. only one event can occur at a time. The set of program states, S, is obtained by taking Cartesian product of domains of all program variables. Each state corresponds to a valuation of all the variables of the program.

The transition relation X can be derived from the set of events, E, of the program. Every event $e \in E$ defines a binary relation on the program's states, $X_e \subseteq S \times S$, such that

$$\begin{aligned} X_e = \{ & ((x_1, x_2, \ldots, x_n), (x'_1, x'_2, \ldots, x'_n)) : \\ & (x_1, x_2, \ldots, x_n) \in \mathbf{enabled}(e) \wedge \\ & ((x_1, x_2, \ldots, x_n), (x'_1, x'_2, \ldots, x'_n)) \in \mathbf{action}(e) \} \end{aligned}$$

where $x_1, x_2, \ldots, x_n$ and $x'_1, x'_2, \ldots, x'_n$ denote values of data and control variables before and after the execution of event e. The sets $\mathbf{enabled}(e)$ and $\mathbf{action}(e)$ respectively denote the enabling condition and action of event e. Hence the global transition relation X is $X = \bigcup_{e \in E} X_e$.

Representing Properties

Consider our safety property, which states that *two processes are never in the critical section at the same time.* This is equivalent to asserting that the following formula is always satisfied over the program's reachable state-space:

$$\neg\ (pc_1 = C_1\ \wedge pc_2 = C_2)$$

We call this type of assertion a *state formula*, which is generated by the following grammar:

$$\begin{array}{rcl} f & ::= & t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists \mathbf{var}\ f \\ t & ::= & (t) \mid t + t \mid \mathbf{var} \mid \mathbf{constant} \end{array}$$

Here, the terminals **constant** and **var** represent integer constants and variables, respectively. Using this base language, we can easily represent formulas including $<$, $=$, $\vee$, $\forall$, as well as multiplication by a constant. The set of closed formulas defined by the above grammar forms the theory of integers with addition, called *Presburger arithmetic*. An important property of Presburger arithmetic is that validity is decidable, i.e., given a closed formula, there are decision procedures which can decide if it is true or false [15, 16].

For a given program, its class of state formulas, SF, is the set of Presburger formulas in which all open variables are the variables of the program. In other words, these are exactly the formulas that have an interpretation via the program's valuation function, L.

We often "close off" a state-formula using its corresponding set representation. For example, for the bakery program,

$$\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\}$$

represents all program states in which both processes are in the critical section at the same time. Also, note that we may also include quantifiers in our state formulas, which allows postulating more complicated properties; e.g.,

$$\{(pc_1, pc_2, a, b) : \exists \alpha \,.\, b = 2\alpha + 1\}$$

$s \models f$	iff	$L(s,f) = \mathsf{True}$, where $f \in SF$
$s \models \neg f$	iff	$s \not\models f$
$s \models f \wedge g$	iff	$s \models f$ and $s \models g$
$s_0 \models \forall \bigcirc f$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models \exists \bigcirc f$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models \forall \Diamond f$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models f$
$s_0 \models \exists \Diamond f$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models f$

Table 1: Semantics of our temporal logic.

represents the set of states in which b is odd.

Our temporal operators extend directly from those of CTL. We use four modal operators as the logic's basis – the "quantified-next-state" operators ($\exists\bigcirc$ and $\forall\bigcirc$), and "quantified-eventuality" operators ($\exists\Diamond$ and $\forall\Diamond$). Thus, the logic we use to reason about a program is generated over the set

$$\{f \in SF,\ \exists\bigcirc,\ \forall\bigcirc,\ \exists\Diamond,\ \forall\Diamond,\ \wedge,\ \vee,\ \neg\}$$

As usual, CTL's quantified-invariant operators can easily be represented as $\exists\Box f = \neg\forall\Diamond\neg f$, and $\forall\Box f = \neg\exists\Diamond\neg f$, respectively.

The semantics of temporal logic formulas is defined on a program's transition system $M = (S, I, X, L)$, as shown in Table 1. The table represents the semantics in terms of a program's *paths*, where a path $(s_0, s_1, s_2, \ldots)$ is a (finite or infinite) sequence of states, such that for each successive pair of states $(s_i, s_{i+1}) \in X$.

Unlike Clarke *et al* [5], we do not require the transition relation X to be total. Rather, our semantics is defined using maximal paths [3] (as opposed to infinite paths). A maximal path is one which is either infinite, or it ends with a state that has no successors.

Consider the two properties of the bakery algorithm we discussed at the beginning of this section. Our safety property – *two processes are never at the critical section at the same time* – can easily be expressed as:

$$\forall\Box(\neg\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\})$$

which is equivalent to

$$\neg(\exists\Diamond\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\}).$$

Note that for the sake of clarity, we are using the symbols C_1 and C_2 as valuations for the control variables pc_1 and pc_2. In practice, we map all program variables to integer domains.

Now recall our progress property: *If a process starts waiting for entry to the critical section, it eventually gets in.* For the first process, this can be expressed as:

$$\forall\Box(\{(pc_1, pc_2, a, b) : pc_1 = W_1\} \rightarrow \forall\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = C_1\}))$$

which is equivalent to:

$$\neg(\exists\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = W_1\} \wedge \neg(\forall\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = C_1\})))).$$

Symbolic Representations

As we have shown, Presburger formulas – and their corresponding set representations – give us a convenient way to symbolically encode sets of states in a program. We can also encode the transition relation of a program with Presburger formulas, as long as we restrict the underlying program's enabling conditions and actions to Presburger formulas. This prevents us, for example, from defining multiplication within a single event.

In general, representing the entire transition relation (X) with a single Presburger formula will not be practical. Since this formula will represent all the program's events, it may end up being quite large. Rather, we exploit individual event-transition relations (the X_e's), which yield a natural decomposition of X. For each $e \in E$, if we assume that **enabled**(e) and **action**(e) are representable as Presburger formulas, then X_e is also representable as a Presburger formula. This results in $|E|$ Presburger formulas, which together symbolically encode the transition relation X.

Representing all of a program's state space (S) by a single Presburger formula may also lead to a very large formula, which will be of little use for analysis. Instead we use the structure of the program as follows:

1. We partition the state-space S, and symbolically represent each partition class individually.
2. We incrementally analyze the program by considering one class at a time.

In general, let $P = \{S_1, S_2, \ldots, S_n\}$ be a partitioning of the program states; i.e.,

$$S = \bigcup_i S_i \quad \text{and} \quad \forall i, j\ ,\ S_i \cap S_j = \emptyset.$$

In our method, partitioning of the program states is based on the valuations of the control variables. We assume that each valuation defines a control point, which in turn induces a partition class. I.e., when applied to the bakery program this yields:

$$P = \{S_{(T_1,T_2)}, S_{(T_1,W_2)}, \ldots, S_{(C_1,C_2)}\}$$

where, for example,

$$S_{(C_1,T_2)} = \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = T_2\}.$$

While the entire state-space can be partitioned via S_i's, so can any subset of S. That is, let $Q \subseteq S$. Then

Operation	Description
$F \cap G$	intersection of F and G
$F \cup G$	union of F and G
$F - G$	difference of F and G
F^{-1}	inverse of F (relation)
$F[G]$	restrict domain of F (relation) to G (set) and return the range of the resulting relation

Table 2: Symbolic operations on Presburger sets and relations.

$P_Q = \{Q_1, Q_2, \ldots, Q_n\}$, where $Q_i = Q \cap S_i$, is a partitioning of Q. E.g., for our bakery program, the set

$$Q = \{(pc_1, pc_2, a, b) : a < b\}$$

denotes all states in which a is less than b. Using a partitioning via control points, we have

$$Q_{(C_1,C_2)} = \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2 \wedge a < b\}.$$

This formula denotes the set of states where a is less than b and both processes are at the critical section.

SYMBOLIC MODEL CHECKER

In the previous section we defined the syntax and semantics of our temporal logic. Now we show how a formula in the logic can be computed symbolically.

For a given temporal formula f, we implicitly use the symbolic form of f to denote the set of states that satisfy f – i.e., the set $\{s | s \models f\}$. From elementary set theory, we then get the following interpretations:

$$f \wedge g = f \cap g \qquad f \vee g = f \cup g \qquad \neg f = S - f$$

To carry these operations out *symbolically*, we use a toolset called the *Omega library* [12], which includes a large collection of object classes to manipulate Presburger formulas. Of particular use to us are the functions shown in Table 2, which are all implemented as Omega functions. These operations take Presburger sets or relations as inputs, and return the symbolic form of a Presburger set or relation as output.

We also use these operators to help us symbolically compute the temporal operators. First, we define a function $\mathbf{pred} : 2^S \to 2^S$ called *predecessor function*, which, given a set of states, returns all the states which can reach this set in one step (i.e. after execution of a single event). Formally,

$$\mathbf{pred}(Q) = \{s : s' \in Q \wedge (s, s') \in X\}$$

or using the notation in Table 2, $\mathbf{pred}(Q) = X^{-1}[Q]$. Since we do not represent X as a single relation, we don't compute $\mathbf{pred}(Q)$ in a single step. Rather, we

PROCEDURE CHECK(f)
(Subformulas are computed recursively.)
CASE
$f \in SF$: RETURN(f)
$f = \neg f_1$: RETURN($S - f_1$)
$f = f_1 \wedge f_2$: RETURN($f_1 \cap f_2$)
$f = \exists \bigcirc f_1$: RETURN($\mathbf{pred}(f_1)$)
$f = \forall \bigcirc f_1$: RETURN($S - \mathbf{pred}(S - f_1)$)
$f = \exists \Diamond f_1$: $Q_0 = f_1$
$Q_{i+1} = Q_i \cup \mathbf{pred}(Q_i)$
RETURN(Q_n) when $Q_n = Q_{n+1}$
$f = \forall \Diamond f_1$: $Q_0 = f_1$
$Q_{i+1} = Q_i \cup ((S - \mathbf{pred}(S - Q_i)) \cap \mathbf{pred}(Q_i))$
RETURN(Q_n) when $Q_n = Q_{n+1}$

Figure 4: Symbolic Model Checker.

compute $\mathbf{pred}$ with respect to our program's partitioning, and compute one formula for each set of the partition. Formally, let $P = \{S_1, S_2, \ldots, S_n\}$ be our partitioning of the program states, and let Q be a set of program states. Since $Q = \bigcup_{S_i \in P}(Q \cap S_i)$, we can compute $\mathbf{pred}(Q)$ as

$$\begin{aligned} &\mathbf{pred}(Q) = \mathbf{pred}(\textstyle\bigcup_{S_i \in P}(Q \cap S_i)) \\ = &\textstyle\bigcup_{S_i \in P} \mathbf{pred}(Q \cap S_i) = \bigcup_{S_i \in P, e \in E} X_e^{-1}[Q \cap S_i] \end{aligned}$$

This is nothing more than symbolically computing the *weakest precondition* of Q with respect to each set of the partition. It often makes sense to do this individually for each partition class, since many formulas inherently refer to only a small number of classes. E.g., let $Q = \{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\}$, i.e. Q is the set of states where both processes are at the critical section. Then,

$$\begin{aligned} \mathbf{pred}(Q) &= \mathbf{pred}(Q \cap S_{(C_1,C_2)}) \\ &= \bigcup_{e \in \{e_{W_1}, e_{W_2}\}} X_e^{-1}[Q \cap S_{(C_1,C_2)}] \end{aligned}$$

which is equal to:

$$\begin{aligned} &\{(pc_1, pc_2, a, b) : pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b = 0 \vee a < b)\} \\ \cup &\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a = 0 \vee b < a)\} \end{aligned}$$

Now, according to the semantics of the temporal operators defined in Table 1, we immediately have

$$\exists \bigcirc f = \mathbf{pred}(f) \qquad \forall \bigcirc f = S - \mathbf{pred}(S - f)$$

So given a symbolic representation for f, we can symbolically compute $\exists \bigcirc f$ and $\forall \bigcirc f$ using the function $\mathbf{pred}$.

As for $\exists\Diamond$ and $\forall\Diamond$, consider the functionals $\tau_{EF} = \lambda y.\ f \vee \exists \bigcirc y$ and $\tau_{AF} = \lambda y.\ f \vee (\forall \bigcirc y \wedge \exists \bigcirc y)$. The least fixpoints of τ_{EF} and τ_{AF} are equal to $\exists\Diamond f$ and $\forall\Diamond f$, respectively [13]. To compute these fixpoints we can use the following property:

Property 1 *For all $n \in \mathcal{Z}$,*

$$\exists\Diamond f \quad \supseteq \quad \bigcup_{i=0}^{n}(\lambda y.\ f \vee \exists \bigcirc y)^i(\mathsf{False})$$
$$\forall\Diamond f \quad \supseteq \quad \bigcup_{i=0}^{n}(\lambda y.\ f \vee (\forall \bigcirc y \wedge \exists \bigcirc y))^i(\mathsf{False})$$

This property tells us that every element in the sequence $\mathsf{False} = \emptyset, \tau_{EF}(\emptyset), \tau_{EF}^2(\emptyset), \tau_{EF}^3(\emptyset), \ldots$ is a subset of the least fixpoint of τ_{EF}; similarly, every element in the sequence $\mathsf{False} = \emptyset, \tau_{AF}(\emptyset), \tau_{AF}^2(\emptyset), \tau_{AF}^3(\emptyset), \ldots$ is a subset of the least fixpoint of τ_{AF}. Since τ_{EF} and τ_{AF} are both monotonic, and since we start the sequence with $\emptyset$, these sequences are non-decreasing. So we can keep iterating until we reach a fixpoint, and when we do, we know that it is the least fixpoint [14].

These methods lead directly to the semi-decision procedure shown in Figure 4. Given a program and a temporal logic formula, the procedure will (attempt to) symbolically compute the set of program states that satisfy the input formula. That is, the procedure will give an exact answer if it converges.

In finite-state systems this is the end of the story, since the procedure is guaranteed to converge. This is also true in some infinite-state systems, depending on the structure of the program and the formula. This is the case in our analysis of the bakery algorithm, where we can compute an exact fixpoint.

Algorithm, Revisited

Recall the mutual exclusion requirement for the bakery algorithm, represented as:

$$\neg(\exists\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\})).$$

To prove this property, the model checker starts by computing the least fixpoint $\exists\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\})$, with the first iterate initialized to $Q_0 = \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\}$. The fixpoint computation converged to a set Q after 4 iterations (for a total computation time of 2.85 seconds on a Sun SPARCstation 5), where Q is partitioned as follows:

$$\begin{array}{rcl}
Q_{(T_1,T_2)} & : & pc_1 = T_1 \wedge pc_2 = T_2 \wedge \mathsf{False} \\
Q_{(T_1,W_2)} & : & pc_1 = T_1 \wedge pc_2 = W_2 \wedge b = 0 \\
Q_{(T_1,C_2)} & : & pc_1 = T_1 \wedge pc_2 = C_2 \wedge b = 0 \\
Q_{(W_1,T_2)} & : & pc_1 = W_1 \wedge pc_2 = T_2 \wedge a = 0 \\
Q_{(W_1,W_2)} & : & pc_1 = W_1 \wedge pc_2 = W_2 \wedge (a = b = 0 \vee \\
 & & a = 0 \wedge 1 \leq b \vee b = 0 \wedge 1 \leq a) \\
Q_{(W_1,C_2)} & : & pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b = 0 \vee a < b) \\
Q_{(C_1,T_2)} & : & pc_1 = C_1 \wedge pc_2 = T_2 \wedge a = 0 \\
Q_{(C_1,W_2)} & : & pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a = 0 \vee b < a) \\
Q_{(C_1,C_2)} & : & pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{True}.
\end{array}$$

Our top-level formula is $\neg(\exists\Diamond\{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\})$, which means that we have to compute $S - Q$. This will yield the set of states which can never reach a violation of the mutual exclusion property. For the bakery algorithm the initial condition is $I = \{(pc_1, pc_2, a, b) : pc_1 = T_1 \wedge pc_2 = T_2 \wedge a = b = 0\}$, and it is easy to see that $I \subseteq (S - Q)$; i.e., all of the initial states satisfy the safety property, and hence the property is proved.

The model checker was also able to prove the progress property:

$$\begin{array}{l}\neg(\exists\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = W_1\} \wedge \\ \qquad \neg(\forall\Diamond(\{(pc_1, pc_2, a, b) : pc_1 = C_1\})))).\end{array}$$

The inner ($\forall\Diamond$) and outer ($\exists\Diamond$) fixpoint computations converge in 9 and 1 iterations, respectively (with a total computation time of 7.64 seconds). Hence, both our requirements were proved by the symbolic model checker described in Figure 4.

In general, however, we will not be so fortunate. After all, we have a Turing-computable language – which means that we may easily keep iterating forever without reaching a fixpoint. (If this were not the case, we would be able to solve the halting problem.) Thus we also need a conservative approximation method, which will converge in finite time.

APPROXIMATION TECHNIQUES

It is usually impossible to determine *in advance* whether an exact fixpoint computation will converge; in general this is itself an undecidable problem. So when we carry out our analysis, we first attempt to compute an exact solution. If the calculation takes an unacceptable amount of time, or perhaps uses too much space, we appeal instead to a conservative approach. A conservative analyzer is one which never yields a "false positive" (and reports that a property holds when in fact it does not), but it may yield a "false negative," and indicate that a property does not hold when it really does.

Ticket Algorithm.

Our exact analyzer diverged when we fed it the so-called *ticket algorithm* [2], along with its related mutual exclusion property. The program text is presented in Figure 5. In particular, note its similarity to the Bakery example. Both algorithms use a few shared variables to maintain mutual exclusion, and they ensure progress in a similar way. Here, when a process attempts to enter to the critical section it first gets a *ticket*, which has a higher number than all of the tickets issued before. The value of the next available ticket is stored in the global variable t, while s holds the highest ticket value served thus far. New tickets are obtained by executing a fetch-and-add on t – where the returned values are stored in either a or b, the local ticket repositories. A cus-

Data Variables: a, b, t, s: integer
Control Variables: $pc_1 : \{T_1, W_1, C_1\}$, $pc_2 : \{T_2, W_2, C_2\}$
Initial Condition: $t = s$
Events:

e_{T_1}	**enabled:**	$pc_1 = T_1$
	action:	$pc_1' = W_1 \wedge a' = t \wedge t' = t + 1$
e_{W_1}	**enabled:**	$pc_1 = W_1 \wedge s \geq a$
	action:	$pc_1' = C_1$
e_{C_1}	**enabled:**	$pc_1 = C_1$
	action:	$pc_1' = T_1 \wedge s' = s + 1$
e_{T_2}	**enabled:**	$pc_2 = T_2$
	action:	$pc_2' = W_2 \wedge b' = t \wedge t' = t + 1$
e_{W_2}	**enabled:**	$pc_2 = W_2 \wedge s \geq b$
	action:	$pc_2' = C_2$
e_{C_2}	**enabled:**	$pc_2 = C_2$
	action:	$pc_2' = T_2 \wedge s' = s + 1$

Figure 5: The Ticket Mutual-Exclusion Algorithm

tomer can enter the critical section when the last-used ticket s catches up to its local ticket number. Again, the mutual-exclusion property is:

$$\neg(\exists\Diamond\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\})$$

which asserts that no two processes can be in the critical section at the same time. When the exact analyzer went to work on this property, it attempted to symbolically enumerate ways that both a and b could be less than s. Since s and t are arbitrary integers, this method failed to converge. At this point we turned to the conservative analyzer, which yielded a positive response.

What is Conservative?
As explained above, the goal of model-checking is to compute a truth set $\{s : s \models f \wedge s \in S\}$ for some temporal formula f. (In the sequel – as in the preceding text – we use the formula f to represent this set.) We also typically wish to determine whether $I \subseteq f$, i.e., whether f holds for the initial states in I. But if we can't directly compute f on our program, the next-best-thing is to generate a lower-bound for f, denoted f^-, such that $f^- \subseteq f$. Then if we determine that $I \subseteq f^-$, we have also achieved our objective – that $I \subseteq f$. However if $I \not\subseteq f^-$, we can conclude nothing.

Consider the property $g = \neg\exists\Diamond(a = b)$. We want to show that there is *no* state on *any* reachable path where $a = b$. Again, the objective is to first compute a lower bound g^- for g, and then check if $I \subseteq g^-$. But note that there is a subtle twist here: Since we seek to carry out our analysis in a compositional (syntax-directed) manner, the most direct way to obtain g^- is to (1) to compute an *upper* approximation h^+ for the term $h = \exists\Diamond(a = b)$, and then (2) to let $g^- = S - h^+$. This follows directly from set theory, since if $h \subseteq h^+$, then $\neg(h^+) \subseteq \neg h$. Hence we need algorithms to compute both lower and upper bounds of temporal formulas.

First we show that in negation-free formulas, lower and upper bound computations are trivially compositional. Recall that model-checking is a recursive procedure (see Figure 4), which works in a bottom-up manner on a formula's structure – first on the leaves (i.e., the state-formulas), then up to the next-level enclosing formula, and so on. Thus, the compositionality of an approximation follows directly from the fact that all operators other than $\neg$ are monotonic. I.e., for any operator $op \in \{\wedge, \vee, \exists\bigcirc, \forall\bigcirc, \exists\Diamond, \forall\Diamond\}$

$$f^- \subseteq f \subseteq f^+ \Longrightarrow op(f^-) \subseteq op(f) \subseteq op(f^+)$$

This means that any lower/upper approximation for a formula can be computed using the corresponding lower/upper approximation for its subformulas.

As for negation, we showed above how we can handle formulas with a single "$\neg$" at the outermost level, by exploiting the properties $(\neg f)^- = \neg(f^+)$ and $(\neg f)^+ = \neg(f^-)$. We can easily generalize this method for the arbitrary use of negation. To compute a bound for a CTL formula f, the following procedure determines which of f's subformulas require an upper bound, and which require a lower bound.

1. Mark the root of the parse tree for formula f with a minus sign ("$-$") if a lower bound is desired, and with a plus sign ("$+$") if an upper bound is desired.
2. Using a preorder tree traversal, visit each node in the tree, mark each node with the mark of its parent, unless its parent is a $\neg$ operator. In that case mark the node with the opposite bound.

Computing Upper Bounds
Suppose our objective is to compute an upper bound for either $\exists\Diamond f$ or $\forall\Diamond f$ for some formula f. In examining Figure 4, we see that the exact iterations for $\exists\Diamond f$ and $\forall\Diamond f$ form increasing sequences over the lattice of state sets $(2^S, \emptyset, S, \cup, \cap)$, with $\emptyset$ as bottom, S as top, $\cup$ as join, and $\cap$ as meet. I.e., we have that

$$Q_0 \subseteq Q_1 \subseteq Q_2 \ \ldots$$

and moreover, each Q_i is a lower approximation to the fixpoint. But since we have an infinite state-space, there may in fact be infinitely many members of the chain, with each member containing successively more states than its preceding neighbor. From elementary fixpoint theory we know that a least fixpoint exists – but it may simply not be computable. Hence our job is to accelerate the computation, and "leap-frog" over multiple of the members of the chain – perhaps at the risk of overshooting the exact least fixpoint. As long as the result is larger than the exact fixpoint, we have an upper approximation.

The way we go about this is as follows. If the exact iteration sequence is $Q_0, Q_1, Q_2, \ldots$, then we find a *majorizing* sequence $\hat{Q}_0, \hat{Q}_1, \hat{Q}_2, \ldots$ such that (1) for each i, $Q_i \subseteq \hat{Q}_i$, and (2) the $\hat{Q}_i$ sequence terminates. From (1) we immediately have that the fixpoint of the $\hat{Q}_i$'s is an upper approximation to the least fixpoint of the Q_i's; from (2) we can obtain it in finite time.

To generate the $\hat{Q}_i$'s, we adopt a method developed by Cousot and Cousot, within the framework of abstract interpretation [8]. That is, we define an operator called widening, or "∇", which majorizes the union computation as follows:

For any pair of sets P, P', $P \cup P' \subseteq P \nabla P'$.

Using a suitable widening operator, we slightly redefine the procedure for $\exists\Diamond f$ from Figure 4:

$$\begin{aligned} \hat{Q}_0 &= f \\ \hat{Q}_{i+1} &= \hat{Q}_i \nabla (\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i)) \\ (\exists\Diamond f)^+ &= \hat{Q}_n \quad \text{when} \quad \hat{Q}_n = \hat{Q}_{n+1} \end{aligned}$$

From the monotonicity of the **pred** operator, one can easily show by induction that this sequence does indeed majorize the Q_i's computed in Figure 4. (We leave the proof to the reader.) And if the sequence terminates, the final iterate is an upper bound for $\exists\Diamond f$. For $\forall\Diamond f$ we use the analogous approximation technique:

$$\begin{aligned} &\hat{Q}_0 = f \\ &\hat{Q}_{i+1} = \hat{Q}_i \nabla (\hat{Q}_i \cup ((S - \mathbf{pred}(S - \hat{Q}_i)) \cap \mathbf{pred}(\hat{Q}_i))) \\ &(\forall\Diamond f)^+ = \hat{Q}_n \quad \text{when} \quad \hat{Q}_n = \hat{Q}_{n+1} \end{aligned}$$

If we reach termination, we have that $(\forall\Diamond f)^+$ is an upper bound for $\forall\Diamond f$.

Our goal is to find a widening operator which (1) yields a suitable (i.e., reasonably tight) upper bound for union, and (2) forces the $\hat{Q}_i$ sequence to converge. To do this, we generalize the application of widening used by Cousot and Halbwachs in [9], where the idea is to "guess" the direction of growth in the checker's Q_i iterates, and to extend the successive iterates in these directions. Cousot and Halbwachs defined a widening operator $\hat{\nabla}$ that accomplishes this for *convex polyhedra* – i.e., regions formed by a conjunction of affine constraints. If both P and P' are convex, then $P \hat{\nabla} P'$ is defined by the constraints in P which are also satisfied by P'. Hence, $P \hat{\nabla} P'$ is built by simply removing constraints from P; since we cannot remove infinitely many constraints, the finiteness property is satisfied.

A simple example shows how widening can be used in the context of our event-action language, and with Presburger sets. Consider the following simple program, which consists of only one event:

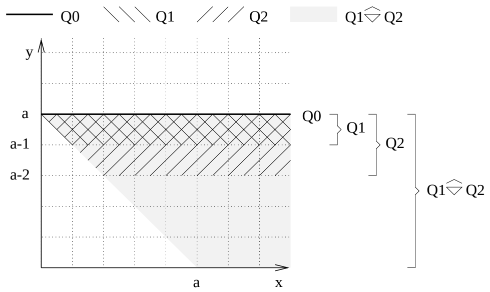

Figure 6: A simple example demonstrating how the widening operator $\hat{\nabla}$ works.

Data Variables: x, y: positive integer
Events:
e **enabled:** $x > 0$
action: $x' = x - 1 \wedge y' = y + 1$

Assume that we wish to check the property $\forall\Box\{(x,y) : y \neq a\}$, where a is a positive constant. Our symbolic model checker will convert this property to $\neg\exists\Diamond(\{(x,y) : y = a\})$, and first try to compute an exact fixpoint for $\exists\Diamond(\{(x,y) : y = a\})$. Figure 6 shows the regions Q_0, Q_1, Q_2 generated by the first iterations of the exact algorithm. At this point, it might be predicted that the sequence will diverge (and indeed, it will). Thus, we can set $\hat{Q}_0 = Q_1$, and then note that $Q_2 = \hat{Q}_0 \cup \mathbf{pred}(\hat{Q}_0)$. We then obtain $\hat{Q}_1$ by computing $Q_1 \hat{\nabla} Q_2$:

$$\begin{aligned} Q_1 \hat{\nabla} Q_2 &= \{(x,y) : a \leq x + y \wedge a - 1 \leq y \leq a\} \\ &\hat{\nabla} \ \{(x,y) : a \leq x + y \wedge a - 2 \leq y \leq a\} \\ &= \{(x,y) : a \leq x + y \wedge 0 \leq y \leq a\} \end{aligned}$$

The iterations converge, since this formula is also generated for $\hat{Q}_2$. When we negate the result, we get $\{(x,y) : x + y < a \vee a < y\}$. In other words, if our initialization of x and y satisfies this condition, then the invariant will indeed hold.

However, a program's state space is not always convex; in fact, most (exact) fixpoint computations are composed of a (potentially large) number of disjuncts, each defining a convex polytope. To accommodate this we generalize the widening definition in [9] to handle multiple polyhedra. Assume that we have two Presburger sets Q and R, where $Q \subseteq R$. Then Q and R can be represented as:

$$Q = q_1 \cup q_2 \cup \ldots \cup q_m \quad \text{and} \quad R = r_1 \cup r_2 \cup \ldots r_m \ldots \cup r_n$$

where all the q_i's and r_i's are convex polytopes, and where $m \leq n$, and for all $1 \leq i \leq m$, $q_i \subseteq r_i$. Then we can define our new widening operator to be

$$Q \nabla R = \bigcup_{i=1}^{n} p_i \qquad (\dagger)$$

such that for $i \leq m$, $p_i = (q_i \widehat{\bigtriangledown} r_i)$, and for $m < i \leq n$, we have $p_i = r_i$.

We do face some technical issues when we use this definition in practice. Assume that we are computing a $\exists\Diamond$ property, and that

$$\hat{Q}_i = q_1 \cup q_2 \cup \ldots \cup q_m$$

where each of the q_j's is convex. Then $\hat{Q}_{i+1} = \hat{Q}_i \bigtriangledown (\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i))$, with

$$\begin{aligned} \hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i)) &= (\textstyle\bigcup_{j=1}^m q_j) \bigcup (\textstyle\bigcup_{j=1}^m \mathbf{pred}(q_j)) \\ &= (q_1 \cup q_2 \cup \ldots \cup q_m) \bigcup (p_1 \cup p_2 \cup \ldots \cup p_n) \end{aligned}$$

Here the p_j's represent a convex decomposition of $\bigcup_{j=1}^m \mathbf{pred}(q_j)$. Now we use a simple algorithm to merge selected q_j's with p_j's in a pairwise fashion. For each q_j $(1 \leq j \leq m)$ we scan the p_k's $(1 \leq k \leq n)$, looking for polyhedra to merge. This is done by computing the convex hull of $q_j \cup p_k$ – denoted $\mathbf{hull}(q_j \cup p_k)$ – and determining if it is equal to $q_j \cup p_k$. If so, we delete the p_k term and replace q_j with $\mathbf{hull}(q_j \cup p_k)$. We continue this process until a maximum amount of merging is accomplished, after which we have:

$$\begin{aligned} \hat{Q}_i &= q_1 \cup q_2 \cup \ldots \cup q_m \\ \hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i) &= r_1 \cup r_2 \cup \ldots \cup r_l \end{aligned}$$

such that $m \leq l$, and for all $1 \leq j \leq m$, $q_j \subseteq r_k$, Then the conditions for $\bigtriangledown$ in (†) are satisfied, and therefore we can use it as our widening operation.

Note, however, that the r_i decomposition of $\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i)$ may include too many terms, since it is possible that there will be little potential for merging the q_i's with the p_k's. So to ensure that we converge, we also assign an upper bound to the number of disjoint convex regions we wish to represent. When we reach this bound we force-merge disjoint regions by replacing them with their convex hull – even if the hull properly contains their union. Since we are computing upper bounds, we can do this at the expense of losing precision.

Computing Lower Bounds

Recall that each iteration of an exact fixpoint computation will yield a lower a bound for $\exists\Diamond f$ and $\forall\Diamond f$. So to obtain a lower approximation for the purposes of analysis, we need only stop after a finite number of iterations; in this manner we are guaranteed to have a conservative approximation. Of course the question is: when do we stop?

Our verifier uses the following rules: if it is handling the outermost formula, then after each iteration it checks whether the initial states are included in the current lower bound. If so it stops, since the property is proved. If not it keeps going. Obviously there will be cases where this method fails to converge, and if this happens the tool will not be able to prove or disprove the property. However, the user is able to interact with the analyzer, and periodically monitor its progress; thus the user can optionally "pull the plug" on waiting for a response.

If the fixpoint we are computing is a subformula of another computation, the analyzer sets an arbitrary time to stop generating an approximation – after which it used in the next-higher formula. But if the analyzer is unable to prove or disprove the outermost formula, the user may optionally return and improve the lower bound by continuing the fixpoint sequence.

Approximate Analysis of the Ticket Algorithm

Now we demonstrate the approximation methods on the Ticket Algorithm(Figure 5). Recall the mutual exclusion property, which is represented in temporal logic as:

$$\neg(\exists\Diamond\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\})$$

Using the negation-labeling algorithm we get

$$(\neg(\exists\Diamond\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\}^+)^+)^-$$

which means that we want to compute an upper bound for $\exists\Diamond\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\}$, i.e., the states which violate mutual exclusion. The symbolic model checker computes the upper bound using the widening technique developed in the previous section; it converges after 9 iterations (with a CPU time of 7.32 seconds). The result is a Presburger set Q, which is partitioned as

$$\begin{aligned} \hat{Q}_{(T_1,T_2)} &: pc_1 = T_1 \wedge pc_2 = T_2 \wedge t < s \\ \hat{Q}_{(T_1,W_2)} &: pc_1 = T_1 \wedge pc_2 = W_2 \wedge t \leq s \\ \hat{Q}_{(T_1,C_2)} &: pc_1 = T_1 \wedge pc_2 = C_2 \wedge t \leq s \\ \hat{Q}_{(W_1,T_2)} &: pc_1 = W_1 \wedge pc_2 = T_2 \wedge t \leq s \\ \hat{Q}_{(W_1,W_2)} &: pc_1 = W_1 \wedge pc_2 = W_2 \wedge (b \leq s \wedge a \leq s \vee \\ &\quad\; t \leq s+1 \wedge b \leq s \vee t \leq s+1 \wedge a \leq s) \\ \hat{Q}_{(W_1,C_2)} &: pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b \leq s \vee t \leq s+1) \\ \hat{Q}_{(C_1,T_2)} &: pc_1 = C_1 \wedge pc_2 = T_2 \wedge t \leq s \\ Q_{(C_1,W_2)} &: pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a \leq s \vee t \leq s+1) \\ Q_{(C_1,C_2)} &: pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{True}. \end{aligned}$$

However, note that we are actually computing $\neg(\exists\Diamond\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1 \wedge pc_2 = C_2\})$, i.e., we have to get $S - Q$, which will give us a lower approximation for the states which respect mutual exclusion. Recall that the ticket algorithm's initial condition is $I = \{(pc_1, pc_2, a, b, s, t) : pc_1 = T_1 \wedge pc_2 = T_2 \wedge t = s\}$, and observe that $I \subseteq (S - Q)$. This means that all the initial states of the program satisfy the safety property, hence the property is proved.

We also wish to prove starvation-freedom, where the relevant formula for process 1 is:

$$\neg(\exists\Diamond(\{(pc_1, pc_2, a, b, t, s) : pc_1 = W_1\} \wedge$$
$$\neg(\forall\Diamond(\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1\})))).$$

The negation-labeling algorithm converts this to:

$$(\neg(\exists\Diamond(\{(pc_1, pc_2, a, b, t, s) : pc_1 = W_1\}^+ \wedge$$
$$(\neg(\forall\Diamond(\{(pc_1, pc_2, a, b, t, s) : pc_1 = C_1\}^-)^-)^-)^+)^+)^+)^-.$$

Note that because of the double negation, the inner fixpoint $\forall\Diamond$ is marked with "–" (i.e., a lower bound), whereas the outer fixpoint $\exists\Diamond$ is marked with "+." The model checker computes the $\forall\Diamond$ property exactly, in 5 fixpoint iterations; hence the lower bound turns out to be exact. Then it computes an upper bound for the $\exists\Diamond$ property in 7 iterations, by using the widening technique (for a total CPU time of 27.03 seconds). After the lower bound for the whole formula is computed, the model checker reports that all the initial states do indeed satisfy the liveness property.

Using Forward Analysis

The fixpoint computations we have described thus far are *backward* techniques, in that they start with a property f, and then use **pred** to determine which states can reach f. The last step is to determine whether the initial condition I is included in the derived set. Alternatively, it may be useful to *start* with I, compute an upper approximation RS^+ to the reachable state-space, and then use RS^+ to help in the model-checking process. We can practically accomplish this by altering the symbolic model checker to restrict its computations to states in RS^+.

To help generate the upper bound RS^+, we define a function $\mathbf{succ} : 2^S \to 2^S$, called the *successor function*, which, given a set of states, returns the states reachable from this set in one step. Formally,

$$\mathbf{succ}(Q) = \{s : s' \in Q \wedge (s', s) \in X\}$$

In particular, note that this is the "forward analogue" to the **pred** function. We claim that the (exact) reachable state space of a program is the least fixpoint of the functional $\lambda y.\ (I \vee \mathbf{succ}(y))$; hence it can be computed using the techniques we previously developed for $\exists\Diamond f$. Moreover, the same upper bound method works as well:

$$\begin{aligned} R_0 &= I \\ R_{i+1} &= R_i \bigtriangledown (R_i \cup \mathbf{succ}(R_i)) \\ (RS)^+ &= R_n \text{ when } R_n = R_{n+1} \end{aligned}$$

After computing RS^+, we restrict the result of every operation in the model checker (Figure 4) to RS^+. In other words, when each iterate Q_i is produced, it is replaced by $Q_i \cap RS^+$. Most importantly, we can also use this technique when we compute approximate *backward* fixpoints, as defined above.

Data Variables: i, b, o_1, o_2: integer		
Constants: a: integer		
Initial Condition: $i = a \wedge (b = o_1 = o_2 = 0)$		
Events:		
e_W	**enabled:**	$i > 0$
	action:	$i' = i - 1 \wedge b' = b + 1$
e_{R_1}	**enabled:**	$b > 0$
	action:	$o_1' = o_1 + 1 \wedge b' = b - 1$
e_{R_2}	**enabled:**	$b > 0$
	action:	$o_2' = o_2 + 1 \wedge b' = b - 1$

Figure 7: A Producer-Consumer Algorithm

We demonstrate this method on a producer-consumer problem. Consider the program in Figure 7, which implements a buffer with one "writer" and two "readers." The writer continues executing until it exhausts its input, while the readers consume data whenever some is available.

The property we wish to prove is as follows. *If the input is exhausted, and the buffer is empty, then the total items consumed is equal to the total number produced*:

$$\forall\Box(\{(i, b, o_1, o_2) : i = b = 0 \to o_1 + o_2 = a\})$$

When we translate this into existential form, and submit to our negation-labeling process, we get:

$$(\ \neg(\exists\Diamond(\neg\{(i, b, o_1, o_2) : i = b = 0 \to o_1 + o_2 = a\}))^+\ )^-$$

Before tackling this property, we first obtain the upper bound RS^+ for the reachable states of the program

$$RS^+ \ = \ \{(i, b, o_1, o_2) : i + b + o_1 + o_2 = a\}$$

The approximate fixpoint computation given above converges to this set in 4 iterations. Now we turn to the correctness property. As explained above, we set the Q_0 seed of the $\exists\Diamond$ fixpoint algorithm to:

$$\neg\{(i, b, o_1, o_2) : i = b = 0 \to o_1 + o_2 = a\}\ \cap\ RS^+$$

which yields the empty set! In other words, the fixpoint computation trivially converges to **False**. When we negate it, we see that all *reachable* program states do indeed satisfy the property we wanted to prove. The total verification effort required a CPU time of 2.87 seconds.

CONCLUSIONS

We have presented a new symbolic model checker for infinite-state programs, which evaluates safety and liveness properties expressed in a variant of CTL. Our method is based on three key concepts:

- Symbolically encoding transition relations and state sets in Presburger formulas, which we can efficiently manipulate using the Omega library [12].
- Partitioning a program's state-space via the control variables, and using the partition classes as repositories for the model checker's formula-labeling computations.
- Approximating fixpoint computations with techniques that guarantee convergence in finite time.

We presented three infinite-state concurrent programs, which demonstrated our three analysis techniques: exact, approximate-backward and approximate-forward analysis. While the programs do not contain many lines of code, they exhibit subtle interplay between the infinite-state variables and predicates controlling execution flow. They are the sort of programs usually analyzed in hand proofs.

There is much work remaining. While our multiple-polyhedra widening approximation helped solve two of the problems in this paper, it can often be rather coarse. In general it sacrifices precision for finite termination. We are currently developing more precise widening methods, which produce abstractions on execution *paths* rather than sets of states. As we acquire more experience with both types of approximations, we hope to determine which techniques work best for different classes of programs, and why.

We also plan to develop a better way to handle compositionality. We currently form our state-partitions over the Cartesian-product of all variable domains. When we scale to large numbers of processes we will obviously need a more compositional approach. To this end, we believe we can use many of the analogous methods developed for finite-state systems.

REFERENCES

[1] R. Alur, T. A. Henzinger, and P. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering 22(3)*, March 1996.

[2] Gregory R. Andrews, Concurrent Programming, Principles and Practice. 1991, The Benjamin/Cummings Publishing Company.

[3] Andre Arnold, Finite Transition Systems: Semantics of Communicating Systems, New Jersey, 1994, Prentice Hall.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, 1990.

[5] 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, (April 1986).

[6] E. M. Clarke, O. Grumberg, D. E. Long Model checking and abstraction. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[7] D. C. Cooper. Programs for mechanical program verification. In *Machine Intelligence 6*, B. Meltzer and D. Michie, editors, pages 43–59, New York, 1971, American Elsevier.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of Colloque International sur la programmation*, April 1976.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual Symposium on Principles of Programming Languages*, 1978, ACM Press.

[10] L. K. Dillon. *ACM Transactions on Programming Languages and Systems*, 12(4):643–669, (October 1990).

[11] S. L. Hantler and J. C. King. An Introduction to proving the correctness of programs. *ACM Computing Surveys* 8(3):331–353, (September 1976).

[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman and D. Wonnacott. The Omega Library (version 1.00) Interface Guide. Available at <http://www.cs.umd.edu/projects/omega>.

[13] D. Kozen Results on the propositional μ-Calculus, *Theoretical Computer Science* 27:333–354, (1983).

[14] K. L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. PhD Thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[15] D. C. Oppen. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic. *Journal of Computer and System Sciences* 16:323–332, (1978).

[16] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–104, (August 1992).

[17] A. Udaya Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3):225–262, (September 1993).

[18] J. M. Wing and M. Vaziri-Farahani. Model Checking Software Systems: A Case Study In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–138, (October 1995).