Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic*

Tevfik Bultan, Richard Gerber and William Pugh

Department of Computer Science
University of Maryland, College Park, MD 20742, USA

Abstract. 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.

1 Introduction

In recent years, there has been a surge of progress in the area of automated analysis for finite-state systems. Several reasons for this success are: (1) the development of powerful techniques such as *model-checking* (e.g., [5, 7]), which can efficiently verify safety and liveness properties; (2) innovative new data structures that symbolically encode large sets of states in compact formats (e.g., [4, 5]); and (3) new ways of carrying out *compositional* and *local* analysis, to assuage the "state explosion" usually associated with concurrency (e.g., [6, 9, 14]). But when transition systems are not restricted to be finite, most of these techniques are no longer applicable, as they inherently depend on all underlying types being bounded. Also general safety and liveness properties become undecidable for infinite transition systems.

We have developed a symbolic model checker to attack this problem, which symbolically encodes transition relations and sets of states using affine constraints on integer variables, logical connectives and quantifiers (i.e., Presburger formulas). Then, it efficiently manipulates these formulas (via a fast Presburger solver called the Omega library [15, 17]) to derive truth sets of temporal logic formulas and their fixpoint computations. Also, we use conservative approximation techniques in analysis of infinite state programs, which guarantee convergence by allowing false negatives.

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

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

Other methods have been proposed to deal with infinite-state programs like these, and we note some of them here. In [8] Clarke *et al.* present a conservative model checking technique, by producing a finite abstraction of the program (e.g. via a congruence relation modulo a suitable integer), and then checking the property of interest on the abstraction. In [12] Dingel and Filkorn extend this method using "assumption-commitment" style reasoning and theorem proving. While these techniques require the user to find the appropriate abstractions – and hence are not completely automatable – we see them as being orthogonal to our approach. There may be cases where abstraction methods can vastly reduce the state space without achieving a finite representation. In these cases our model checker can be used on the infinite abstract models.

Our work was influenced by known techniques from abstract interpretation [10, 11]; specifically, we use some approximation methods first developed for that domain. Most reachability properties can be formulated as least fixpoints over sets of a program's states; if the state space is infinite, these fixpoints may not be computable. Abstract interpretation provides a way of approximating these fixpoints via a technique known as "widening" – which can compute a least fixpoint's upper bound in finite time. Since our basic temporal operators require similar computations, we were able to successfully use this method in conjunction with the Omega library.

Finally, our encoding of program states is similar to that used by Alur *et al.* in verifying hybrid systems [1]. A hybrid system is a discrete control automaton, which interacts with continuously-changing, external parameters. Like us, Alur *et al.* used an application of widening to help solve verification queries over linear hybrid automata – in which transition relations are defined in terms of affine constraints over the variables of the system.

The fundamental difference in our work 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 express and prove properties such as "x *is even*," using quantification. In general, satisfiability problems over constraints with integer variables are significantly harder to deal with. For example, checking to see if there exists an integer solution to a set of linear constraints is NP-hard, while the analogous real-valued problem can be solved in polynomial time. Also, we take our fixpoints by storing, at each step, unions of convex regions (with possible stride constraints); Alur *et al.* force intermediate results into a single convex region. While their strategy is one way to control potential state explosion, we have found that in our problem domain, most interesting properties cannot be proved unless multiple convex regions are used at each point.

The paper is organized as follows. First 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 *conservative approximations*, we discuss the specific approximation techniques for computing upper and lower bounds of fixpoints. Finally, we conclude with some discussion on our results.

Data Variables: a, b: positive integer					
Control Variables: $pc_1 : \{T_1, W_1, C_1\}$, $pc_2 : \{T_2, W_2, C_2\}$					
Initial Condition: $a = b = 0 \wedge pc_1 = T_1 \wedge pc_2 = T_2$					
Events:					
e_{T_1}	**enabled:**	$pc_1 = T_1$	e_{T_2}	**enabled:**	$pc_2 = T_2$
	action:	$pc_1' = W_1 \wedge a' = b + 1$		**action:**	$pc_2' = W_2 \wedge b' = a + 1$
e_{W_1}	**enabled:**	$pc_1 = W_1 \wedge$ $(a < b \vee b = 0)$	e_{W_2}	**enabled:**	$pc_2 = W_2 \wedge$ $(b < a \vee a = 0)$
	action:	$pc_1' = C_1$		**action:**	$pc_2' = C_2$
e_{C_1}	**enabled:**	$pc_1 = C_1$	e_{C_2}	**enabled:**	$pc_2 = C_2$
	action:	$pc_1' = T_1 \wedge a' = 0$		**action:**	$pc_2' = T_2 \wedge b' = 0$

Fig. 1. The bakery algorithm.

2 Representation of Programs and Properties

We use the event-action language from [18] as our syntax for concurrent programs, with a semantics defined in terms of infinite transition systems. A concurrent program $C = (V, I, E)$ is represented by (1) a finite set of data and control variables V; (2) an initial condition I, which specifies the starting states of the program; and (3) a finite set of events E, where each event is considered atomic. The state of a program is determined by the values of its data and control variables. We assume that the domain of each variable is a countable set. 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 variables of the program.

Consider the concurrent program shown in Figure 1, which implements the *bakery algorithm* [2] to achieve mutual exclusion between two processes. Here the control points for each process are denoted T, W, C, which stand for *thinking, waiting* or *in critical section*, respectively. If a variable v is used in an event, then the symbol v' denotes the new value of v after the action. If v is not mentioned in the action of an event, then we assume that its value is not altered by that event.

When a process wants to enter the critical section it first gets a ticket, which will be higher than those of all other processes currently in the critical section or waiting for entry. In the above system, 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. Note that variables a and b can increase without bound; i.e., this is not a finite-state program.

Given a program $C = (V, I, E)$ in the above language, we model it as an infinite transition system $M = (S, I, X, L)$, where S is the set of states, I is the set of initial states, $X \subseteq S \times S$ is the transition relation (derived from the set of events E), and $L : S \times SF \rightarrow \{\mathsf{True}, \mathsf{False}\}$ is the valuation function for state formulas over the program's variables. (We define the set of state formulas SF below.) The set of states S is obtained by taking Cartesian product of domains of all program variables; hence each state corresponds to a valuation of all the variables 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 $X_e = \{(s, s') : s \in \mathbf{enabled}(e) \wedge (s, s') \in \mathbf{action}(e)\}$ where s and s' denote program's states before and after the execution of event e, respectively. The sets $\mathbf{enabled}(e)$ and $\mathbf{action}(e)$ respectively denote the enabling condition and action of event e. Hence the global transition relation is $X = \bigcup_{e \in E} X_e$. 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.

Presburger formulas. The bakery algorithm's mutual exclusion requirement asserts that the following property stays invariant over all executions: $\neg\,(pc_1 = C_1 \wedge pc_2 = C_2)$. We call this type of assertion a *state formula*. And in general, we define the set of state formulas SF for a program C as all Presburger formulas which range over program's variables. Presburger formulas are generated by the following grammar:

$$f \ ::= \ t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists \mathbf{var}\ f \qquad t \ ::= \ (t) \mid t + t \mid \mathbf{var} \mid \mathbf{constant}$$

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.

In general, the worst-case time bound for determining validity in Presburger arithmetic is prohibitive [13]. Yet we have found that the Omega library [15, 17] is quite efficient at solving the problems that arise in our analysis, which typically possess a small number of constraints, and do not contain multiple levels of alternating quantifiers. The Omega library uses extensions of Fourier variable elimination to solve integer programming problems, along with a set of transformation functions and heuristics to help convert real-valued approximations into discrete-valued solutions.

Temporal Properties. We use four CTL-style modal operators as the basis for our temporal logic – 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, 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 a temporal formula is defined on the paths of a program's transition system, $M = (S, I, X, L)$. 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.* [7], we do not require the transition relation X to be total. Rather, the 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. The semantics of the temporal operators can then be defined as follows: A state s_0 satisfies $\forall \bigcirc f$ ($\exists \bigcirc f$) if and only if for all (some) maximal paths $(s_0, s_1, s_2, \ldots)$ with length ≥ 2, s_1 satisfies f. A state s_0 satisfies $\forall\Diamond f$ $(\exists\Diamond f)$ if and only if for all (some) maximal paths $(s_0, s_1, s_2, \ldots)$ there exists an i such that s_i satisfies f.

In this language, the bakery algorithm's mutual-exclusion property is expressed as $\forall\Box(\neg(pc_1 = C_1 \wedge pc_2 = C_2))$, that is, the two processes never reach the critical section at the same time.

3 Symbolic Analysis

Presburger formulas – and their corresponding set-theoretic interpretations – give us a convenient way to symbolically encode sets of program states. Moreover, we also use this encoding to represent the program's underlying transition relation. For a given event e, if we assume that **enabled**(e) and **action**(e) are both representable as Presburger formulas (which prevents us, for example, from defining multiplication within a single event), then X_e is representable as a Presburger formula. This results in $|E|$ Presburger formulas, which together symbolically encode the transition relation X.

To carry out our analysis, we exploit the natural partitioning induced by valuations of the control variables, and we incrementally analyze the program by considering one class at a time. When applied to the bakery program this yields the following partitioning of the state space: $P = \{S_{(T_1,T_2)}, S_{(T_1,W_2)}, \ldots, S_{(C_1,C_2)}\}$ where, for example,

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

We can then partition any subset of S as follows: If $Q \subseteq S$, then $P_Q = \{Q_1, Q_2, \ldots, Q_n\}$ is a partitioning of Q, where each $Q_i = Q \cap S_i$ (for all $S_i \in P$). E.g., in the 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 $P_Q = \{Q_{(T_1,T_2)}, Q_{(T_1,W_2)}, \ldots, Q_{(C_1,C_2)}\}$ where, for example,

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

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

After partitioning of the state space, we use the Omega library [15] to help *symbolically* compute the truth sets for the temporal properties at hand. The Omega library includes a large collection of object classes to efficiently manipulate Presburger formulas; to date it has mainly been used in high-performance compilers, specifically for dependence analysis, program transformations, and detecting redundant synchronization [16, 17]. The particular Omega functions we use are shown in Figure 2(A). These functions take symbolic representations of sets or relations as inputs (i.e., a Presburger formula representing a set or a relation), and return the symbolic form of a set or a relation as output.

To symbolically compute the temporal operators, we define a function **pred** : $2^S \rightarrow 2^S$, called the *predecessor function*, which, given a set of states, returns

Symbolic Omega Operations	
$F \cap G$	: symbolic intersection
$F \cup G$	: symbolic union
$F - G$	: symbolic difference
F^{-1}	: symbolic inverse of relation F
$F[G]$	: restrict domain of relation F to constraint G and return the range of the result
$\mathbf{hull}(F)$	: convex hull of F

Procedure Check(f)	
Case	
$f \in SF$	: Return(f)
$f = \neg f_1$	: Return($S - f_1$)
$f = f_1 \wedge f_2$	: Return($f_1 \cap f_2$)
$f = f_1 \vee f_2$	: Return($f_1 \cup f_2$)
$f = \exists \bigcirc f_1$	: 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 $Q_n = Q_{n+1}$
$f = \forall \Diamond f_1$	: $Q_0 = f_1$
$Q_{i+1} =$	$Q_i \cup (\mathbf{pred}(Q_i) - \mathbf{pred}(S - Q_i))$
	Return(Q_n) when $Q_n = Q_{n+1}$

Fig. 2. (A) Omega functions, and (B) symbolic model checker.

all the states that can reach this set in one step (i.e. after execution of a single event):

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

Using the Omega operator in Figure 2(A) we have $\mathbf{pred}(Q) = X^{-1}[Q]$. Moreover, we can symbolically compute **pred** with respect to our program's partitioning, and maintain a formula for each partition class, as follows:

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

By performing this computation individually for each partition class, we exploit the fact that many formulas inherently involve only small parts of the program's state space. For example, consider the states where both processes are at the critical section, or $Q = \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\}$. Then we have

$$\begin{aligned}\mathbf{pred}(Q) = & \bigcup_{e \in \{e_{W_1}, e_{W_2}\}} X_e^{-1}[Q \cap S_{(C_1, C_2)}] \\ = & \{(pc_1, pc_2, a, b) : pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b = 0 \vee a < b)\} \\ & \cup \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a = 0 \vee b < a)\}.\end{aligned}$$

Now, given a symbolic representation for a set f, we can symbolically compute $\exists \bigcirc f$ and $\forall \bigcirc f$ using **pred**, as follows:

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

As for $\exists\Diamond$ and $\forall\Diamond$, consider the functionals $\tau_{\exists\Diamond f} = \lambda y.\ f \vee \exists \bigcirc y$ and $\tau_{\forall\Diamond f} = \lambda y.\ f \vee (\forall \bigcirc y \wedge \exists \bigcirc y)$. The least fixpoints of $\tau_{\exists\Diamond f}$ and $\tau_{\forall\Diamond f}$ are equal to $\exists\Diamond f$ and $\forall\Diamond f$, respectively. Using well-known properties from lattice theory, it can be shown that every element in the sequence $\mathsf{False} = \emptyset$, $\tau_{\exists\Diamond f}(\emptyset)$, $\tau^2_{\exists\Diamond f}(\emptyset)$, $\tau^3_{\exists\Diamond f}(\emptyset), \ldots$, is a subset of the least fixpoint of $\tau_{\exists\Diamond f}$; similarly, every element in the sequence $\mathsf{False} = \emptyset$, $\tau_{\forall\Diamond f}(\emptyset)$, $\tau^2_{\forall\Diamond f}(\emptyset)$, $\tau^3_{\forall\Diamond f}(\emptyset)$, ..., is a subset of the least fixpoint of $\tau_{\forall\Diamond f}$. So when these monotonically increasing sequences reach a fixpoint, we know that it is the least fixpoint.

These methods lead directly to the semi-decision procedure shown in Figure 2(B) (subformulas are computed recursively). Given a program and a temporal logic formula, the model checker will (attempt to) symbolically compute the set of program states that satisfy the input formula – and the procedure will yield an exact answer if it converges.

Bakery Algorithm, Revisited. Recall the mutual exclusion requirement for the bakery algorithm, which is equivalent to: $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$. To compute the least fixpoint $\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, the model checker initialized the first iterate to $Q_0 = \{(pc_1, pc_2, a, b) : pc_1 = C_1 \wedge pc_2 = C_2\}$. After 4 iterations, the fixpoint computation converged to a set Q (for a total computation time of 2.85 seconds on a Sun SPARCstation 5), where Q is partitioned as follows:

$$\boxed{\begin{array}{ll}
Q_{(T_1,T_2)}\ : pc_1 = T_1 \wedge pc_2 = T_2 \wedge \mathsf{False} & Q_{(T_1,C_2)} : pc_1 = T_1 \wedge pc_2 = C_2 \wedge b = 0 \\
Q_{(T_1,W_2)} : pc_1 = T_1 \wedge pc_2 = W_2 \wedge b = 0 & Q_{(C_1,T_2)} : pc_1 = C_1 \wedge pc_2 = T_2 \wedge a = 0 \\
Q_{(W_1,T_2)} : pc_1 = W_1 \wedge pc_2 = T_2 \wedge a = 0 & Q_{(C_1,C_2)} : pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{True} \\
Q_{(W_1,C_2)}\ : pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b = 0 \vee a < b) & \\
Q_{(C_1,W_2)}\ : pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a = 0 \vee b < a) & \\
\multicolumn{2}{l}{Q_{(W_1,W_2)} : pc_1 = W_1 \wedge pc_2 = W_2 \wedge (a = b = 0 \vee a = 0 \wedge 1 \le b \vee b = 0 \wedge 1 \le a)}
\end{array}}$$

Since the top-level formula is $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, the model checker computes $S - Q$. Then it checks if $I \subseteq (S - Q)$ and concludes that all of the initial states satisfy the safety property, hence the property is proved.

The model checker also proved the starvation freedom property, $\forall\Box(pc_1 = W_1 \rightarrow \forall\Diamond(pc_1 = C_1))$, which is equivalent to $\neg\exists\Diamond(pc_1 = W_1 \wedge \neg\forall\Diamond(pc_1 = C_1))$. The inner ($\forall\Diamond$) and outer ($\exists\Diamond$) fixpoint computations converged in 9 and 1 iterations, respectively (with a total computation time of 7.64 seconds).

4 Approximation Techniques

Since we have a Turing-computable language, our exact model-checker in Figure 2(B) may keep iterating forever without reaching a fixpoint. Thus we also need a conservative approximation method, which will always converge. 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.

Indeed, our exact analyzer diverged when we fed it the so-called *ticket algorithm* [2], along with its related mutual exclusion property (see Figure 3). In particular, note its similarity to the bakery algorithm. The difference is that the value of the next available ticket is stored in the global variable t, while another global variable s holds the highest ticket value served thus far. New tickets are obtained by executing a fetch-and-add on t. A customer can enter the critical section when the last-used ticket s catches up to its local ticket number.

When the exact analyzer went to work on the mutual exclusion property of the ticket algorithm, it attempted to symbolically enumerate ways that both a

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: $t = s \wedge pc_1 = T_1 \wedge pc_2 = T_2$
Events:

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

Fig. 3. The ticket mutual-exclusion algorithm.

and b could be less than s. Since s and t are unbounded, this method failed to converge.

4.1 What is Conservative?

If we cannot directly compute a property f for a 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 cannot conclude anything.

Since we seek to carry out our analysis in a recursive manner (as in the exact analyzer in Figure 2(B)), we have to compute an approximation to a formula by first computing approximations for its subformulas. Hence, with a property like $g = \neg h$, we first need to compute an *upper* approximation h^+ for the subformula h, and then let $g^- = S - h^+$.

When analyzing a negation-free formula, the compositionality of an approximation follows directly from the fact that all operators other than "$\neg$" are monotonic. This means that any lower/upper approximation for a formula can be computed using the corresponding lower/upper approximation for its subformulas. As for handling arbitrary levels of negation, we can easily generalize the above mentioned method for outermost negation operators. That is, to approximate a temporal 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.

4.2 Computing Upper Bounds

When the algorithms in Figure 2(B) attempt to compute fixpoints for $\exists\Diamond$ and $\forall\Diamond$, they may generate sequences of increasing lower bounds which never converge. And 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 members of the chain – perhaps at the risk of over-shooting 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 reaches a fixpoint after finitely many iterates. Thus the fixpoint of the $\hat{Q}_i$'s is an upper approximation to the least fixpoint of the Q_i's.

To generate the $\hat{Q}_i$'s, we currently adopt a method developed by Cousot and Cousot, within the framework of abstract interpretation [10]. That is, we define an operator called widening, or "$\bigtriangledown$", which majorizes the union computation as follows: For any pair of sets P, P', $P \cup P' \subseteq P \bigtriangledown P'$. Using a suitable widening operator, we can redefine the procedures for $\exists\Diamond f$ and $\forall\Diamond f$ from Figure 2(B) as:

$\hat{Q}_0 = f$	$\hat{Q}_0 = f$
$\hat{Q}_{i+1} = \hat{Q}_i \bigtriangledown (\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i))$	$\hat{Q}_{i+1} = \hat{Q}_i \bigtriangledown (\hat{Q}_i \cup (\mathbf{pred}(\hat{Q}_i) - \mathbf{pred}(S - \hat{Q}_i)))$
$(\exists\Diamond f)^+ = \hat{Q}_n$ when $\hat{Q}_n = \hat{Q}_{n+1}$	$(\forall\Diamond f)^+ = \hat{Q}_n$ when $\hat{Q}_n = \hat{Q}_{n+1}$

From the monotonicity of the **pred** operator, one can easily show by induction that these sequences do indeed majorize the Q_i's computed in Figure 2(B). And the final iterates are upper bounds for $\exists\Diamond f$ and $\forall\Diamond f$.

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 $\hat{Q}_i$ sequences to converge. In defining our widening operator, we generalized a technique used by Cousot and Halbwachs in [11]. The idea is to "guess" the direction of growth in the model-checker's Q_i iterates, and to extend the successive iterates in these directions. Cousot and Halbwachs' widening operator $\widehat{\bigtriangledown}$ does this for *convex polyhedra* – i.e., regions formed by a conjunction of affine constraints. If both P and P' are convex, then $P \widehat{\bigtriangledown} P'$ is defined by the constraints in P which are also satisfied by P'. For example,

$$\{(x,y) : x-1 \leq y \leq x\} \;\widehat{\bigtriangledown}\; \{(x,y) : x-2 \leq y \leq x\} = \{(x,y) : y \leq x\}$$

Intuitively, if a constraint of P is not satisfied by P' this means that the iterates are increasing in that direction. By removing that constraint we extend the iterates in the direction of growth as much as possible without violating other constraints. Since $P \widehat{\bigtriangledown} P'$ is built by simply removing constraints from P and since we cannot remove infinitely many constraints, the finiteness property is satisfied. *But* because it folds all arguments into a single convex region, a direct application of this method failed to work for us. The reason is that on all of our examples to date, all fixpoint computations were composed of a (potentially large) number of disjuncts, each defining a convex polytope. To accommodate this we generalized $\widehat{\bigtriangledown}$ 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 $Q = q_1 \cup q_2 \cup \ldots \cup q_m$ and $R = r_1 \cup r_2 \cup \ldots \cup r_m \cup \ldots \cup r_n$, 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

$$Q \bigtriangledown R = \bigcup_{i=1}^{n} p_i \text{ s. t. } \forall i \; [i \leq m \rightarrow p_i = q_i \widehat{\bigtriangledown} r_i \text{ and } m < i \leq n \rightarrow p_i = r_i] \quad (\dagger)$$

So, 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

$$\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i) \;=\; (\bigcup_{j=1}^{m} q_j) \bigcup (\bigcup_{j=1}^{m} \mathbf{pred}(q_j)) = (q_1 \cup \ldots \cup q_m) \bigcup (p_1 \cup \ldots \cup p_l)$$

Here the p_k's ($1 \leq k \leq l$) represent a convex decomposition of $\bigcup_{j=1}^{m} \mathbf{pred}(q_j)$. To form the necessary r_i's, we use a simple algorithm to merge selected q_j's ($1 \leq j \leq m$) with p_k's ($1 \leq k \leq l$) in a pairwise fashion. For each q_j ($1 \leq j \leq m$) we scan the p_k's ($1 \leq k \leq l$), looking for polyhedra to merge. This is done by invoking an Omega function to compute 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:

$$\hat{Q}_i = q_1 \cup q_2 \cup \ldots \cup q_m \quad \text{and} \quad \hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i) = r_1 \cup r_2 \cup \ldots \cup r_n$$

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

Note that the r_j decomposition of $\hat{Q}_i \cup \mathbf{pred}(\hat{Q}_i)$ may include too many terms if there is little potential for merging the q_j's with the p_k's. To ensure convergence, 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 that loses precision (which is valid since we are computing upper bounds).

4.3 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 a (user specified) time limit to stop generating an approximation – after which it is 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. Using the negation labeling algorithm, the mutual exclusion property of the ticket algorithm is rendered as $(\neg(\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)^+)^+)^-$. The temporal operator $\exists\Diamond$ is marked with "+" which means that we need an upper bound for the set of states violating mutual exclusion. The upper bound $\hat{Q}$ is computed using the multi-polyhedra widening technique in 9 iterations (with a CPU time of 7.32 seconds). However, since we are actually computing $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, the model checker computes $S - \hat{Q}$, which is a lower approximation for the states which respect mutual exclusion. Then, shows that $I \subseteq (S - \hat{Q})$.

We also wish to prove starvation-freedom. Negation-labeling converts process 1's relevant formula to: $(\neg(\exists\Diamond((pc_1 = W_1)^+ \wedge (\neg(\forall\Diamond(pc_1 = C_1)^-)^-)^+)^+)^+)^-$. 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, it reports that all the initial states do indeed satisfy the liveness property.

5 Remarks

We have presented a new symbolic model checker for infinite-state programs, which evaluates safety and liveness properties. We demonstrated our method on two example programs. While they 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 one 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 methods for reachability properties using transitive closure computation techniques for Presburger formulas [16]. 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 investigate compositional approaches. 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, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
2. G. R. Andrews. Concurrent Programming, Principles and Practice. The Benjamin/Cummings Publishing Company, 1991.
3. A. Arnold. Finite Transition Systems: Semantics of Communicating Systems. Prentice Hall, 1994.
4. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
5. 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 *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 428–439, 1990.
6. T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counter-examples. In *Proc. 1996 Int. Symp. on Software Testing and Analysis*, pages 224–238, 1996.
7. 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.
8. E. M. Clarke, O. Grumberg, D. E. Long Model checking and abstraction. In *Proc. 18th Annual ACM Symp. on Principles of Programming Languages*, pages 343–354, 1992.
9. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. 4th Annual IEEE Symp. on Logic in Computer Science*, pages 464–475, 1989.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Annual ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th Annual ACM Symp. on Principles of Programming Languages*, pages 84–97, 1978.
12. J. Dingel, and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Proc. 7th Int. Conference on Computer Aided Verification, LNCS 939, pages 54–69, 1995.
13. M. J. Fischer and M. O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. *SIAM-AMS Proc.*, Volume 7, pages 27-41, 1974.
14. P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. Ph.D. Thesis, Universite De Liege, 1994.
15. 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>.
16. W. Kelly, W. Pugh, E. Rosser and T. Shpeisman. Transitive closure of infinite graphs and its applications. Technical Report CS-TR-3457, UMIACS-TR-95-48, Department of Computer Science, University of Maryland, 1994.
17. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–104, 1992.
18. A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.