Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results[*][†]

Tevfik Bultan, Richard Gerber and William Pugh
Department of Computer Science
University of Maryland, College Park, MD 20742, USA
{bultan,rich,pugh}@cs.umd.edu

Abstract

Model checking is a powerful technique for analyzing large, finite-state systems. In an infinite-state 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.

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., [6, 8]), 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., [5, 6, 25]); and (3) new ways of carrying out *compositional* and *local* analysis, to assuage the "state explosion" usually associated with concurrency (e.g., [7, 10, 18]). But when transition systems are not restricted to be finite, most of these techniques are no longer applicable, as they inherently depend on all underlying types being bounded.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 is based on the following key concepts:

- Symbolically encoding transition relations and sets of states using affine constraints on integer variables, logical connectives and quantifiers (i.e., Presburger formulas).

---

[*] Preliminary results from this paper appeared as an extended abstract in the Proceedings of the 9th Conference on Computer Aided Verification (CAV '97).

[†] This work was supported in part by ONR grant N00014-94-10228, NSF CCR-9619808 and a Packard Fellowship.

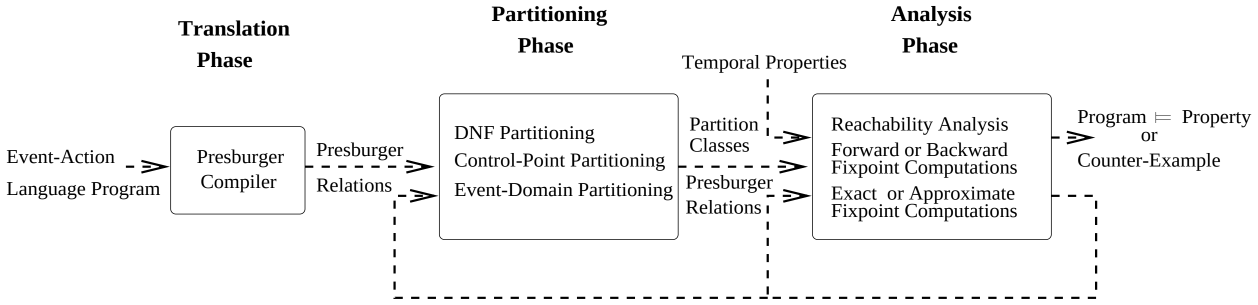

Figure 1: Overview of the Analysis Tool.

- Efficiently manipulating these formulas (via a fast Presburger solver called the Omega library [21, 28]) to derive truth sets of temporal logic formulas and their fixpoint computations.
- Using conservative approximation techniques in analysis of infinite state programs, which guarantee convergence by allowing false negatives.

In any computer system variables are eventually mapped to finite representations. Thus, it might be argued that integer variables can be given a finite range, and programs can then be analyzed as finite-state machines – for example, using BDD's [6, 25]. For several 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. 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.

Given a finite but very large transition system, an analysis technique which has a worst case complexity proportional to the size of the input transition system will run out of resources (i.e., memory) when the worst case is realized. Complexity analysis of model checking for all interesting temporal logics shows that worst case complexity is at least linear in the size of the input transition system. I.e., although model checking finite state systems is decidable (as opposed to infinite state systems) for very large systems it can be intractable. Hence, from a practical stand point, our techniques for analyzing infinite state systems can be viewed as techniques for analyzing very large finite state systems which do not rely on the finiteness of the state space.

In this paper we demonstrate our model checker's effectiveness on some classical infinite-state programs taken from the concurrency literature [3, 30]. While relatively small, they possess some interesting subtleties, especially in the tricky way their infinite-state variables influence control flow.

A summary of our approach is shown in Figure 1, and it depicts how we structure the three main phases: *translation*, *partitioning* and *analysis*.

In the *translation phase*, the system accepts as input a program written in a simple event action language; then it produces the corresponding set of Presburger relations. These Presburger relations are just a symbolic encoding of the program's underlying transition relation.

In the *partitioning phase*, the program's states and transitions are segregated into a set of partition classes, with the objective of reducing the complexity of verification. The motivation here is easy to understand: Consider treating a complex program as a single relation, which might contain multiple if-then-else branches and loops. Then consider pushing through a nontrivial weakest-precondition transform – say, to automatically derive a loop-invariant. In most cases, the result will explode into an unmanageable (potentially infinite) number of constraints. It is true that the general form of our verification problem is undecidable; hence, we cannot eliminate this issue entirely. However, partitioning lets us damp some of its effects – and more importantly, it increases the precision of our conservative techniques. The "default" partitioning strategies are:

- *DNF Partitioning:* This method decomposes the program's transition relation into a disjunctive-normal-form; then pre- and post-condition transforms are carried out for each disjunct, one at a time.
- *Control-Point Partitioning:* This method symbolically decomposes the state-space into partition classes, based on valuations of selected variables (we call these *control-points*).
- *Event-Domain Partitioning:* This method is a finer-grained version of Control-Point Partitioning. It forms its partition classes based on the enabling conditions of events so that in each partition class an event is either enabled for all states or disabled for all states.

We describe these strategies in the following sections. However, we stress two points here: (a) they are used to segregate the program (and its states) into large regions, which distinguish the truth or falsity of crucial temporal properties; and (b) these decompositions do not alter the underlying transition relation of the original program; hence, they preserve all temporal properties.

In the *analysis phase*, verification procedures are applied to help prove (or disprove) properties of interest. Many of these are inter-dependent; i.e., they automatically "call" each other as "subroutines" to obtain the original goal. Nonetheless, many high-level strategy decisions are left to the user. In the sequel, we demonstrate situations in which human input can help; specifically, we show a set of programs (and requirements) which helped reveal the relative strengths and weaknesses of each approach. While some programs were easily verified with certain strategies, the same strategies diverged when used on other programs. We note, however, that the user need not be a model-checking "expert" *per se*; rather, the type of experience we consider useful is more like that a programmer draws on when setting compiler-optimization levels.

As for the techniques, all of the "top-level" techniques we report are *at least* conservative, and several are exact. Some programs can, in fact, be quickly verified using exact symbolic model-checking algorithms, similar to those presented in [25] for BDDs. In our domain, the analogue of a CTL "atomic proposition" is postulated over the Presburger logic, and is propagated through the program as such. We supply both *backward* and *forward* procedures for exact analysis. (The backward version starts with the desired goal, and performs recursive pre-condition transforms to get back to the initial states; the forward procedure works in the opposite fashion.) Additionally, we export conservative variants of the backward and forward techniques, which can obtain lower and upper approximations to the exact fixpoint. Two of the methods use a generalized form of "widening" technique discussed in [13] for abstract interpretation. We demonstrate how the approximate methods work together (with minimal user intervention); i.e., how our driver program integrates lower and upper bounds for sub-formulas, and then derives a conservative result for the high-level goal.

Finally, we describe two procedures which can be run as a pre-processing step before model-checking. These will often accelerate convergence for *all* of the fixpoint techniques, both approximate and exact. One technique derives an upper bound for the program's reachable state space; the other is a related function which approximates the transitive closure of the transition relation. Both methods are guaranteed to converge, and both are used for the following purposes: (1) to reduce the sizes of the Presburger formulas generated during model-checking; and (2) to increase the precision of approximate analysis.

The remainder of the paper is organized as follows. In the following section we overview some related work in the field. Then, we present the syntax and semantics for concurrent programs and their properties. After introducing the Presburger encodings and partitioning techniques, 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. Then, we present a variety of different example programs, and show how the various methods performed on them. Finally, we conclude with some discussion on our results.

2 Related Work

Other methods have been proposed to deal with infinite-state concurrent programs, and we note some of them here. In [9], 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 [16], 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.

Another approach is to use symbolic execution technique [15, 19], 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 [15]. Although this method has the benefit of incrementally proving correctness (as opposed to generating all possible interleavings), it relies on users to come up with the correct assertions.

Our work has some historical antecedents. Cooper [11] 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 Presburger arithmetic. The decision procedure used by Cooper was computationally very expensive which made the validity checks intractable. Also proving correctness as a 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 [12, 13]; 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.

Our recursive approximation technique for temporal properties with nested temporal operators is similar to the one used by Kelb *et al.* in [20]. In [20] a temporal property expressed in mu-calculus is computed conservatively using two abstractions of the same program. One abstraction over estimates the behavior of the program and used for computing universal properties. Another abstraction underestimates the behavior of the program and is used for computing existential properties. Together they can be used to conservatively approximate any mu-calculus property. Similar methods are also used in [14, 27] where an approximation for a temporal formula expressed in mu-calculus is computed using lower or upper approximations of its subformulas. Our approximation techniques are based on approximating the fixpoint computations instead of approximating the program as a whole using abstractions. It is possible to use both of these techniques together.

Finally, our encoding of program states is similar to that used by Alur *et al.* in verifying hybrid systems [1, 2]. 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.

A fundamental difference between our work and the work of Alur *et al.* is that we encode sets of integers – as opposed to the real numbers used in hybrid systems – thus, we can use Presburger

Program: Unbounded Buffer

Data Variables: p, c, q_1, q_2: positive integer

Control Variables: $pc : \{Idle, Send\}$

Initial Condition: $p = c = q_1 = q_2 = 0$

Events:

e_{IS}	**enabled:**	$pc = Idle$
	action:	$pc' = Send$
e_S	**enabled:**	$pc = Send$
	action:	$p' = p + 1 \wedge \ (q_1' = q_1 + 1 \ \vee \ q_2' = q_2 + 1)$
e_{SI}	**enabled:**	$pc = Send$
	action:	$pc' = Idle$
e_R	**enabled:**	$q_1 > 0 \vee q_2 > 0$
	action:	$(q_1 > 0 \wedge q_1' = q_1 - 1 \wedge c' = c + 1) \vee$ $(q_2 > 0 \wedge q_2' = q_2 - 1 \wedge c' = c + 1)$

Figure 2: Example (UB) – Indexing for Two Unbounded Buffers.

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.

3 Programs and Properties

We use the event-action language from [30] 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 [30]. 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 (UB) shown in Figure 2, which handles the counters for two unbounded buffers. Note that there is a single control variable pc which is used to attenuate the producer – and it can either be idle (i.e., $pc = Idle$) or ready-to-send (i.e., $pc = Send$). Alternatively, the consumer has only one event (e_R), which can execute whenever outstanding data is sitting in either of the buffers.

Four data variables are used to keep track of the the data items, from the perspective of the producer, the consumer and the two buffers. The total number of items produced (over the program's lifetime) is kept in the integer p; similarly, the number of items consumed is counted in variable c. The other two variables – q_1 and q_2 – keep track of the outstanding items in the buffers. Note that all the data variables can increase without bound, i.e., this program is not a finite-state program.

In our event-action language, 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.)

Three properties of interest are as follows:

- (UB1) The total number of items produced is equal to the number of items consumed, plus the number of items sitting in the two buffers.
- (UB2) If the sender is idle, the total number of items in the buffers monotonically decreases.
- (UB3) The number of items consumed never exceeds the number produced.

Given scale of the program, one need not possess extraordinary skills to understand that the properties are indeed true. Yet, while we may *know* that *this* program is correct, its infinite state space will overcome most contemporary automated analysis engines. Hence, it is easy to see how a larger, more complex program would, in fact, be inaccessible to a *both* a programmer's (informal) "what-if" checks, *and* to an automated finite-state analyzer.

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 the Cartesian product of domains of all program variables, i.e., given a program with n variables $V = \{v_1, v_2, \ldots, v_n\}$, each state $s \in S$ corresponds to a valuation of all the variables of the program

$$s \equiv \bigwedge_{i=1}^{n} v_i = x_i$$

where $x_i \in \mathbf{domain}(v_i)$.

Every event $e \in E$ defines a binary relation on the program's states, $X_e \subseteq S \times S$, interpreted as follows: If $(s, s') \in X_e$, then s and s' denote program's states before and after the execution of event e, respectively. We define the domain and range of an event e as:

$$\mathbf{domain}(e) = \{\ s \mid \exists (s, s')\ :\ (s, s') \in X_e\} \qquad \mathbf{range}(e) = \{\ s' \mid \exists (s, s')\ :\ (s, s') \in X_e\}$$

The global transition relation is $X \equiv \bigvee_{e \in E} X_e$. 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. Recall requirement (UB3) above, which asserts that the following property stays invariant over all executions: $p \geq c$. 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 the Presburger formulas generated by the following grammar:

$$f \ ::= \ t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists\ \textbf{intvar}\ (f) \qquad t \ ::= \ (t) \mid t + t \mid \textbf{intvar} \mid \textbf{intcons}$$

Here, the terminals **intcons** and **intvar** 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. Hence, given a state formula $f \in SF$ and a program state s we can decide if $s \models f$ by substituting the values of program variables in state s to corresponding open variables in formula f and using a Presburger decision procedure (this is equivalent to checking if $s \wedge f$ is satisfiable).

In general, the worst-case time bound for determining validity in Presburger arithmetic is prohibitive, with a deterministic upper bound of $2^{2^{2^{pn}}}$ (for some constant $p > 1$) [26], and a nondeterministic lower bound of $2^{2^{cn}}$ (for some constant $c > 0$) [17], where n denotes the length of the formula. Yet we have found that the Omega library [21, 28] 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 the temporal logic called Computation Tree Logic (CTL) [8] to specify properties of programs. Four CTL operators form the basis of our logic: quantified-next-state operators $\exists\bigcirc$ and $\forall\bigcirc$, and quantified-until operators $\exists\mathcal{U}$ and $\forall\mathcal{U}$. Thus, the logic we use to reason about a program is generated over the set

$$\{\ f \in SF\ ,\ \exists\bigcirc\ ,\ \forall\bigcirc\ ,\ \exists\mathcal{U}\ ,\ \forall\mathcal{U}\ ,\ \wedge\ ,\ \vee\ ,\ \neg\ \}.$$

As usual, quantified-eventuality ($\exists\Diamond$ and $\forall\Diamond$) and quantified-invariant ($\exists\Box$ and $\forall\Box$) operators can be represented as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \exists\Diamond f & \equiv & \mathsf{True}\ \exists\mathcal{U}\ f & \exists\Box f & \equiv & \neg(\mathsf{True}\ \forall\mathcal{U}\ \neg f) \\ \forall\Diamond f & \equiv & \mathsf{True}\ \forall\mathcal{U}\ f & \forall\Box f & \equiv & \neg(\mathsf{True}\ \exists\mathcal{U}\ f) \end{array}$$

We define the semantics of our CTL temporal operators 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.* [8], we do not require the

$s \models f$	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 $s \models g$
$s \models f \vee g$	iff	$s \models f$ or $s \models g$
$s_0 \models \exists \bigcirc f$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models \forall \bigcirc f$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models f\ \exists\mathcal{U}\ g$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models g$, and for all $j < i$, $s_j \models f$.
$s_0 \models f\ \forall\mathcal{U}\ g$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models g$, and for all $j < i$, $s_j \models f$.

Table 1: Semantics of our temporal logic.

transition relation X to be total. Rather, the semantics is defined using maximal paths [4] (as opposed to infinite paths). A maximal path is one which is either infinite, or ends with a state that has no successors. The semantics of the temporal operators can then be defined on a program's transition system $M = (S, I, X, L)$, as shown in Table 1.

If all the initial states of a program satisfy a temporal property, then we say that the program itself satisfies the temporal property. Formally, given a temporal formula f and a transition system $M = (S, I, X, L)$, $M \models f$ only if $\forall s \in I\ :\ s \models f$.

Based on the temporal logic defined above, we can specify the properties of the unbounded-buffer program as follows:

- (UB1) $\forall\Box(p = c + q_1 + q_2)$: The total number of items produced is equal to the number of items consumed, plus the number of items sitting in the two buffers.

- (UB2) $\forall\Box((pc = \mathit{Idle}\ \wedge\ q_1 + q_2 = i) \rightarrow \forall \bigcirc (q_1 + q_2 \leq i))$: If the sender is idle, the number of items in the buffers monotonically decreases.

- (UB3) $\forall\Box(p \geq c)$: The number of items consumed never exceeds the number produced.

Note that the variable i in (UB2) is not a program variable; rather, it is an open integer variable. This points out one of the strengths of our approach – since i can be treated as a symbolic constant within a temporal expression. The interpretation is that i is bound by a universal quantifier (and our Presburger solver treats it as such).

4 Symbolic Representations

Presburger formulas – and their corresponding set-theoretic interpretations – give us a convenient, compact way to symbolically encode large sets of program states and transitions. For our purposes, the benefit of the Presburger encoding is often realized via the arithmetic inequality operators – which we use to implicitly describe large, nontrivial portions of a program's state space. For example, consider the following formula:

$$p - c \leq q_1 + q_2 \leq p \ \wedge \ p, c, q_1, q_2 \geq 0$$

In geometric terms, the constraints represent all points in an unbounded, 4-dimensional polytope. Yet in terms of our example, the shape corresponds to all of the program's reachable states (minus the program-counter).

Similar gains are realized for transitions. Recall that events are described in terms of affine constraints. (This prevents us from using multiplication in a single event, and ensures that single-step image computations are decidable.) Hence, for a given event e, the transition relation of event e, X_e, is representable as a Presburger formula. So using $|E|$ Presburger formulas, one for each event, we can symbolically encode the transition relation X as

$$X \equiv \bigvee_{e \in E} X_e.$$

The fundamental challenge in the Presburger formula encoding is to keep the sizes of the formulas small during the fixpoint computations. Our symbolic manipulator stores Presburger formulas in a disjunctive form where each disjunct corresponds to a convex region in the program's state space. Since each fixpoint iteration is essentially a pre- or post-condition computation, the number of convex regions may increase very quickly in the presence of if-then-else branches and loops.

Given a Presburger formula in a disjunctive form, our minimization procedures try to merge the disjuncts (i.e., convex regions) to reduce the size of the Presburger representation. Assume that we represent each fixpoint iterate using one Presburger formula. After a couple of fixpoint iterations such a representation would generate a Presburger formula with a large number of convex regions. Trying to merge each pair of convex regions one by one would be computationally very expensive. Instead, we use partitioning heuristics. We partition the state space of the program so that the states with similar properties are placed to the same partition class. Then, we represent each fixpoint iterate using one Presburger formula per partition class. This reduces the complexity of the minimization procedures by segregating convex regions which are unlikely to be merged.

We also use approximation techniques to limit the growth of fixpoint iterates. Since the temporal properties we analyze are undecidable, the fixpoint iterations are not guaranteed the converge. Using conservative approximation techniques, we increase the convergence rate of the fixpoint computations. Partitioning heuristics can increase the precision of these approximation techniques. For example one approximation technique could be to merge two convex regions approximately using an upper or a lower bound. Such a technique is more likely to succeed if the merged regions contain states with similar characteristics.

Below we describe our partitioning heuristics. First we describe transition relation partitioning which enables us to compute fixpoint iterations incrementally using one partition class at a time. Then we present our state-space partitioning strategies.

4.1 Transition Relation Partitioning

The most obvious partitioning of the transition relation, X, comes immediately from the event syntax. That is, we can simply perform all transition computations using one event at a time, akin to the way a program's control-points are used in data flow analysis. However, when events contain multiple disjuncts (e.g., from if-then-else constructs), this method proves insufficient for our purposes – since it still generates too many constraints when we carry out automated analysis. In these cases we require a finer-grained technique.

DNF Decomposition. Given two programs $C_1 = (V, I, E_1)$ and $C_2 = (V, I, E_2)$, with the same variables and initial states, assume the following equivalence holds:

$$\bigvee_{e \in E_1} X_e \equiv \bigvee_{e \in E_2} X_e$$

Then C_1 and C_2 are semantically equivalent, since they satisfy exactly the same set of temporal formulas. However, due to their structural differences (and to the methods used in model-checking), one program may be significantly more computationally expensive to analyze than the other. Hence, it often makes sense to transform E_1 into some type of normal-form E_2 – especially when the normal-form has, on average, demonstrated more efficient analysis results.

Such is the case for *disjunctive normal form* (DNF), i.e., where

$$X \equiv \bigvee_{e \in E} X_e, \qquad X_e \equiv \bigwedge_i c_i^e$$

and where each c_i^e is a single affine constraint over primed and unprimed program variables in V. I.e., each X_e corresponds to a convex polytope in $S \times S$.

Clearly, any transition relation X can be so represented – it is simply a matter of converting X into DNF, and then considering each disjunct as a separate event. For example, Figure 3 shows the DNF decomposition of our running program from Figure 2, where for the sake of conciseness, we do not differentiate between an event's "enabled part" and its "action part" (since this is given implicitly by the primed and unprimed variables). Note that the transformation simply split the separate disjuncts from the original events e_S and e_R, and renamed them as separate events.

4.2 State Space Partitioning

Let P be a partitioning of a program's state-space S, where

$$P = \{S_1, S_2, \ldots, S_p\}, \qquad S \equiv \bigvee_{i=1}^{p} S_i, \qquad i \neq j \Rightarrow\ S_i \wedge S_j \equiv \emptyset.$$

$$
\begin{array}{|l|}
\hline
e_{IS} : pc = \mathit{Idle} \ \wedge \ pc' = \mathit{Send} \\
e_{S1} : pc = \mathit{Send} \ \wedge \ p' = p + 1 \ \wedge \ q_1' = q_1 + 1 \\
e_{S2} : pc = \mathit{Send} \ \wedge \ p' = p + 1 \ \wedge \ q_2' = q_2 + 1 \\
e_{SI} : pc = \mathit{Send} \ \wedge \ pc' = \mathit{Idle} \\
e_{R1} : q_1 > 0 \ \wedge \ q_1' = q_1 - 1 \ \wedge \ c' = c + 1 \\
e_{R2} : q_2 > 0 \ \wedge \ q_2' = q_2 - 1 \ \wedge \ c' = c + 1 \\
\hline
\end{array}
$$

Figure 3: DNF Event-Decomposition.

Then if $Q \subseteq S$ is a subset of program states, we can decompose Q via our partitioning P of S:

$$Q \equiv \bigvee_{i=1}^{p} q_i, \quad \text{where} \quad q_i \equiv Q \wedge S_i.$$

This technique will only prove worthwhile where each S_i unites states with common characteristics (that is, when key formulas are true either for all the states in S_i or none of the states in S_i), and when P distinguishes between large regions of S possessing different characteristics.

Below we present two techniques which can often create such a partitioning, and which are two of the "default options" in our model-checker.

Control-Point Partitioning. Given a program $C = (V, I, E)$, assume we partition the variables into two classes:

$$V = V_1 \cup V_2 \quad \text{s.t.} \quad V_1 \cap V_2 = \emptyset.$$

The valuations for V_1 (or V_2) induce a natural partitioning of S. Letting $V_1 = \{v_{11}, \ldots, v_{1m}\}$, equivalence classes are defined simply as:

$$S_{(x_{11}, \ldots, x_{1m})} \ \equiv \ \bigwedge_{j=1}^{m} v_{1j} = x_{1j} \quad \text{s.t.} \quad x_{1j} \in \mathbf{domain}(v_{1j}).$$

Of course, if V_1 contains many variables, or if their domains are large, we may end up with a huge number of classes; hence the partition variables have to be chosen with care.

Fortunately, many programs yield a natural choice for V_1 – the control points – the number of which are usually far fewer than state valuations. To enable this type of partitioning, our event language makes a syntactic distinction between control and data variables (while semantically, they are considered the same). When our preprocessor translates event-action language code into events, the control variables are isolated as such, which then allows a default partitioning according to their values.

When applied to the program in Figure 2, this strategy yields $V_1 = \{pc\}$ and the following two partition classes (shown pictorially in Figure 4):

$$S_{\mathit{Idle}} \equiv pc = \mathit{Idle} \qquad S_{\mathit{Send}} \equiv pc = \mathit{Send}$$

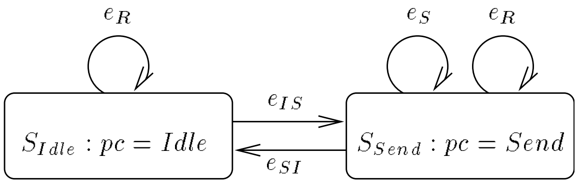

Figure 4: Control-Point Partitioning.

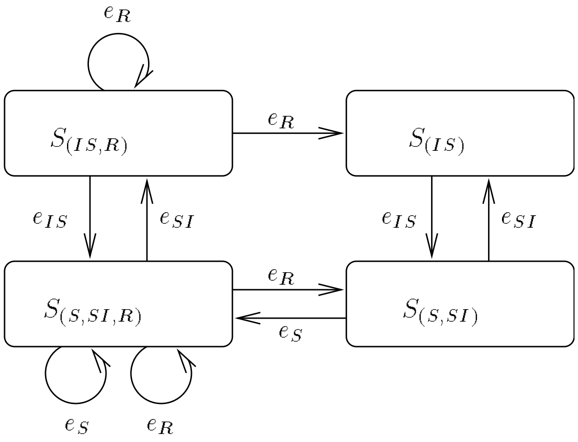

Figure 5: Event-Domain Partitioning.

and so, our partitioning is $P = \{S_{Idle}, S_{Send}\}$. Now consider a formula $Q \equiv p > c$, representing all states in which more items were produced than consumed. The formula is partitioned as $P_Q = \{Q_{Idle}, Q_{Send}\}$, where,

$$Q_{Idle} \equiv pc = Idle \wedge p > c \qquad Q_{Send} \equiv pc = Send \wedge p > c$$

with $Q \equiv Q_{Idle} \vee Q_{Send}$.

Event-Domain Partitioning. The method above can also be generalized to the "implicit control points" inherent in the raw events themselves. In essence, an event's natural control point is just its enabling condition. And, using the enabling conditions as partitioning criterion can yield a much finer-grained decomposition. (This is obviously true when no control variables are isolated in a program.) In this partitioning, each class denotes a region of S in which a specific set of events are enabled to fire. So, given $C = (V, I, E)$, we define its event-domain partitioning P as:

$$P = \{ \ S_i \mid \forall e \in E \ : \ (S_i \wedge \mathbf{domain}(e)) \equiv S_i \ \vee \ (S_i \wedge \neg\mathbf{domain}(e)) \equiv S_i \ \}$$

While this technique yields potentially $2^{|E|}$ partition classes, note that most of these partition classes will probably be empty. The reason is that most of the events will typically have mutually exclusive enabling conditions.

SYMBOLIC OMEGA OPERATIONS

$F \wedge G$: symbolic intersection
$F \vee G$: symbolic union
$\neg F$: symbolic complement
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

Figure 6: Symbolic Omega Functions.

This is certainly the case when we apply the technique to our running example from Figure 2. Event-domain partitioning yields only 4 feasible partition classes (shown pictorially in Figure 5), which is significantly less than $2^{|E|} = 2^4 = 16$. These 4 partition classes are as follows (where each partition class is indexed by the events enabled in the class):

$$\begin{array}{lll@{\qquad}lll} S_{(IS,R)} & \equiv & pc = \mathit{Idle} \wedge (q_1 > 0 \vee q_2 > 0) & S_{(S,SI)} & \equiv & pc = \mathit{Send} \wedge q_1 = 0 \wedge q_2 = 0 \\ S_{(S,SI,R)} & \equiv & pc = \mathit{Send} \wedge (q_1 > 0 \vee q_2 > 0) & S_{(IS)} & \equiv & pc = \mathit{Idle} \wedge q_1 = 0 \wedge q_2 = 0 \end{array}$$

Note that if we have had used the DNF event partitioning first (from Figure 3), we would have produced a finer partitioning with more classes.

5 Symbolic Model-Checker

After generating our symbolic representations in terms of Presburger formulas, we use the Omega library [21] 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 [22, 28, 29]. The particular Omega functions we use are shown in Figure 6. 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 $\mathbf{pre} : 2^S \to 2^S$, called the *precondition function*, which, given a set of states, returns all the states that can reach this set in one step (i.e. after execution of a single event):

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

Using the Omega operator in Figure 6 we have $\mathbf{pre}(Q) \equiv X^{-1}[Q]$. Moreover, we can symbolically compute **pre** with respect to our program's partitioning, and maintain a formula for each partition class, as follows:

$$\mathbf{pre}(Q) \;\equiv\; \mathbf{pre}(\bigvee_{S_i \in P} (Q \;\wedge\; S_i)) \;\equiv\; \bigvee_{S_i \in P} \mathbf{pre}(Q \;\wedge\; S_i) \;\equiv\; \bigvee_{S_i \in P, e \in E} X_e^{-1}[Q \wedge S_i]$$

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 unbounded buffer example – and specifically, the states where producer is *Idle* and buffers are empty:

$$Q \equiv pc = \mathit{Idle} \wedge q_1 = 0 \wedge q_2 = 0.$$

Using control point partitioning, we have

$$\begin{aligned}\mathbf{pre}(Q) \;&\equiv\; \bigvee_{e \in \{e_r, e_{s2}\}} X_e^{-1}[Q \;\wedge\; S_{Idle}] \\ &\equiv\; pc = \mathit{Idle} \wedge (q_1 = 1 \wedge q_2 = 0 \vee q_2 = 1 \wedge q_1 = 0) \vee \; pc = \mathit{Send} \wedge q_1 = 0 \wedge q_2 = 0\end{aligned}$$

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

$$\exists \bigcirc f \equiv \mathbf{pre}(f) \qquad \text{and} \qquad \forall \bigcirc f \equiv \neg\mathbf{pre}(\neg f).$$

As for $\exists\mathcal{U}$ and $\forall\mathcal{U}$, consider the following functionals:

$$\mathcal{F}_{f_1 \exists\mathcal{U} f_2} \equiv \lambda y \;.\; f_2 \vee (f_1 \wedge \exists \bigcirc y) \qquad \text{and} \qquad \mathcal{F}_{f_1 \forall\mathcal{U} f_2} \equiv \lambda y \;.\; f_2 \vee (f_1 \wedge \forall \bigcirc y \wedge \exists \bigcirc y).$$

The least fixpoints of $\mathcal{F}_{f_1 \exists\mathcal{U} f_2}$ and $\mathcal{F}_{f_1 \forall\mathcal{U} f_2}$ are equal to $f_1 \;\exists\mathcal{U}\; f_2$ and $f_1 \;\forall\mathcal{U}\; f_2$, respectively. We have the following result from elementary lattice theory:

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

$$\begin{aligned} f_1 \;\exists\mathcal{U}\; f_2 \;&\supseteq\; \bigvee_{i=0}^{n} (\lambda y \;.\; f_2 \vee (f_1 \wedge \exists \bigcirc y))^i(\mathsf{False}) \equiv \bigvee_{i=0}^{n} \mathcal{F}^i_{f_1 \exists\mathcal{U} f_2}(\mathsf{False}) \\ f_1 \;\forall\mathcal{U}\; f_2 \;&\supseteq\; \bigvee_{i=0}^{n} (\lambda y \;.\; f_2 \vee (f_1 \wedge \forall \bigcirc y \wedge \exists \bigcirc y))^i(\mathsf{False}) \equiv \bigvee_{i=0}^{n} \mathcal{F}^i_{f_1 \forall\mathcal{U} f_2}(\mathsf{False}) \end{aligned}$$

where, given a functional $\mathcal{F}$, $\mathcal{F}^i(f)$ is defined as:

$$\mathcal{F}^i(f) \stackrel{\text{def}}{=} \underbrace{\mathcal{F}(\mathcal{F}(\ldots\mathcal{F}(}_{i \text{ times}} f))\ldots).$$

By the monotonicity of $\mathcal{F}_{f_1 \exists\mathcal{U} f_2}$ and $\mathcal{F}_{f_1 \forall\mathcal{U} f_2}$, we get

$$\bigvee_{i=0}^{n} \mathcal{F}^i_{f_1 \exists\mathcal{U} f_2}(\emptyset) \equiv \mathcal{F}^n_{f_1 \exists\mathcal{U} f_2}(\emptyset) \qquad \text{and} \qquad \bigvee_{i=0}^{n} \mathcal{F}^i_{f_1 \forall\mathcal{U} f_2}(\emptyset) \equiv \mathcal{F}^n_{f_1 \forall\mathcal{U} f_2}(\emptyset).$$

```
PROCEDURE CHECK($f$)
  CASE
    $f \in SF$                       :  RETURN($f$)
    $f = \neg f_1$                   :  RETURN($\neg f_1$)
    $f = f_1 \land f_2$              :  RETURN($f_1 \land f_2$)
    $f = f_1 \lor f_2$               :  RETURN($f_1 \lor f_2$)
    $f = \exists \bigcirc f_1$       :  RETURN($\mathbf{pre}(f_1)$)
    $f = \forall \bigcirc f_1$       :  RETURN($\neg\mathbf{pre}(\neg f_1)$)
    $f = f_1\ \exists\mathcal{U}\ f_2$  :  $Q_0 \equiv f_2$    $Q_{i+1} \equiv Q_i \lor (f_1 \land \mathbf{pre}(Q_i))$
                                        RETURN($Q_n$) when $Q_n \equiv Q_{n+1}$
    $f = f_1\ \forall\mathcal{U}\ f_2$  :  $Q_0 \equiv f_2$    $Q_{i+1} \equiv Q_i \lor (f_1 \land \mathbf{pre}(Q_i) \land (\neg\mathbf{pre}(\neg Q_i)))$
                                        RETURN($Q_n$) when $Q_n \equiv Q_{n+1}$
```

Figure 7: Symbolic Model Checker.

Then using Property 1, it can be shown that every element in the sequence $\mathsf{False} \equiv \emptyset$, $\mathcal{F}_{f_1 \exists\mathcal{U} f_2}(\emptyset)$, $\mathcal{F}^2_{f_1 \exists\mathcal{U} f_2}(\emptyset)$, $\mathcal{F}^3_{f_1 \exists\mathcal{U} f_2}(\emptyset), \ldots$, is a subset of the least fixpoint of $\mathcal{F}_{f_1 \exists\mathcal{U} f_2}$; similarly, every element in the sequence $\mathsf{False} \equiv \emptyset$, $\mathcal{F}_{f_1 \forall\mathcal{U} f_2}(\emptyset)$, $\mathcal{F}^2_{f_1 \forall\mathcal{U} f_2}(\emptyset)$, $\mathcal{F}^3_{f_1 \forall\mathcal{U} f_2}(\emptyset), \ldots$, is a subset of the least fixpoint of $\mathcal{F}_{f_1 \forall\mathcal{U} f_2}$. Since $\mathcal{F}_{f_1 \exists\mathcal{U} f_2}$ and $\mathcal{F}_{f_1 \forall\mathcal{U} f_2}$ are both monotonic, and since we start the sequence with $\emptyset$, these sequences are non-decreasing. When these monotonically increasing sequences reach a fixpoint, we know that it is the least fixpoint [25].

These methods lead directly to the model checking procedure shown in Figure 7 (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. Note that this procedure is a partial-function, i.e., it is not guaranteed to terminate.

We analyzed the running example given in Figure 2 using control point partitioning and the exact model checking procedure given in Figure 7. Recall that one of the requirements for this program was $\forall\Box(p = c + q_1 + q_2)$ which is equivalent to: $\neg(\mathsf{True}\ \exists\mathcal{U}\ (p \neq c + q_1 + q_2))$ or $\neg\exists\Diamond(p \neq c + q_1 + q_2)$. (Although our basic temporal operators are $\exists\mathcal{U}$ and AU, when possible we will use their $\exists\Diamond$ and $\forall\Diamond$ equivalents for clarity of presentation). To compute the least fixpoint $\exists\Diamond(p \neq c + q_1 + q_2)$, the model checker initialized the first iterate to $Q_0 \equiv p \neq c + q_1 + q_2$. The fixpoint computation trivially converged after one iteration to Q, where Q is partitioned as follows:

$$Q_{Idle} \equiv pc = Idle \land p \neq c + q_1 + q_2 \qquad\qquad Q_{Send} \equiv pc = Send \land p \neq c + q_1 + q_2$$

Our top-level formula is $\neg\exists\Diamond(p \neq c + q_1 + q_2)$; hence, the model checker computes $\neg Q$. This yields the set of states which can never reach a violation of the assertion $p = c + q_1 + q_2$. Since the initial condition for the program is $I \equiv p = 0 \land c = 0 \land q_1 = 0 \land q_2 = 0$, it is easy to see that $I \subseteq \neg Q$ (i.e., all of the initial states satisfy the safety property). Hence, the model checker reports that the property is proved (for a total computation time of 0.88 seconds on a Sun SPARCstation 5). The exact model checker also successfully proved the property $\forall\Box((pcs = Idle \wedge q_1 + q_2 = i) \rightarrow \forall \bigcirc (q_1 + q_2 \leq i))$ in 0.77 seconds.

6 Approximation Techniques

Since we have a Turing-computable language, our exact model-checker in Figure 7 may keep iterating forever without reaching a fixpoint. Thus, we also need a conservative approximation method, which will always converge. We define a conservative analyzer as one which always terminates and never yields a spurious result, but may not be able to produce a definite answer in certain cases.

Indeed, our exact analyzer diverged when we fed the unbounded-buffer program, along with its other requirement: $\forall\Box(p \geq c)$. When the exact analyzer went to work on this property, it attempted to symbolically enumerate ways that p could be less than c. Since c is unbounded, this method failed to converge. But as we show in the sequel, the same property was easily proved using a conservative approximation.

6.1 Conservative Analysis

If we cannot directly compute a property f for a program C, 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 $I \subseteq f$, (i.e., we proved that $C \models f$). However, if $I \not\subseteq f^-$, we cannot conclude anything because it can be a *false negative.* In that case we can compute a lower bound for the negated property: $(\neg f)^-$. If we can find a state s such that $s \in I \cap (\neg f)^-$, then we can generate a counter example which would be a *true negative.* If both cases fail, i.e., both $I \not\subseteq f^-$ and $I \cap (\neg f)^- \equiv \emptyset$, then the analyzer can not report a definite answer.

Since we seek to carry out our analysis in a recursive manner (as in the exact analyzer in Figure 7), we have to compute an approximation to a formula by first computing approximations for its subformulas. Hence, to compute a lower bound to a property like $g \equiv \neg h$, we first need to compute an *upper* approximation h^+ for the subformula h, and then let $g^- \equiv S - h^+$. This follows directly from set theory, since $(\neg f)^- \equiv \neg(f^+)$ and $(\neg f)^+ \equiv \neg(f^-)$. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas.

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 negation free 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.

6.2 Computing Upper Bounds with Widening Technique

When the algorithms in Figure 7 attempt to compute fixpoints for $\exists\mathcal{U}$ and $\forall\mathcal{U}$, they may generate sequences of increasing lower bounds which never converge. 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 [12]. That is, we define an operator called widening, or "$\bigtriangledown$", which majorizes the union computation as follows: For any pair of sets P, P', $P \vee P' \subseteq P \bigtriangledown P'$. Using a suitable widening operator, we can compute an upper bound for $f_1\ \exists\mathcal{U}\ f_2$ as:

$$\begin{aligned}
\hat{Q}_i &\equiv \begin{cases} Q_i & \text{if } \ 0 \leq i \leq s \\ \hat{Q}_{i-1} \bigtriangledown (\hat{Q}_{i-1} \vee (f_1 \ \wedge \ \mathbf{pre}(\hat{Q}_{i-1}))) & \text{if } \ i > s \end{cases} \\
(f_1\ \exists\mathcal{U}\ f_2)^+ &\equiv \hat{Q}_n \text{ when } \hat{Q}_n \equiv \hat{Q}_{n+1}
\end{aligned}$$

where $Q_0, Q_1, Q_2, \ldots$, is the sequence generated by the procedure for $f_1\ \exists\mathcal{U}\ f_2$ in Figure 7, and s is the *seed* of the widening sequence. From the monotonicity of the **pre** operator, one can easily show by induction that iterates of this sequence do indeed majorize the Q_i's computed in Figure 7. And when this sequence terminates, the final iterate is an upper bound for $f_1\ \exists\mathcal{U}\ f_2$. For the $f_1\ \forall\mathcal{U}\ f_2$ we can generate a similar majorizing sequence as:

$$\begin{aligned}
\hat{Q}_i &\equiv \begin{cases} Q_i & \text{if } \ 0 \leq i \leq s \\ \hat{Q}_{i-1} \bigtriangledown (\hat{Q}_{i-1} \vee (f_1 \ \wedge \ \mathbf{pre}(\hat{Q}_{i-1}) \wedge (\neg\mathbf{pre}(\neg\hat{Q}_{i-1})))) & \text{if } \ i > s \end{cases} \\
(f_1\ \forall\mathcal{U}\ f_2)^+ &\equiv \hat{Q}_n \text{ when } \hat{Q}_n \equiv \hat{Q}_{n+1}
\end{aligned}$$

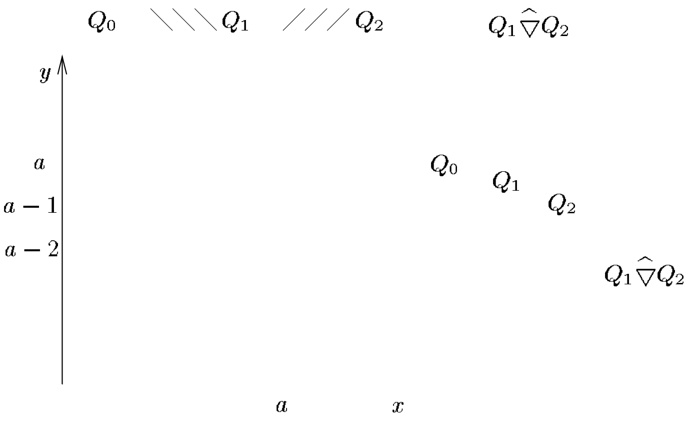

Figure 8: A simple example demonstrating how the widening operator $\widehat{\nabla}$ works.

where $Q_0, Q_1, Q_2, \ldots$, is the sequence generated by the procedure for $f_1 \ \forall\mathcal{U}\ f_2$ in Figure 7.

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 [13]. 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 $\widehat{\nabla}$ does this for *convex polyhedra* – i.e., regions formed by a conjunction of affine constraints. If both P and P' are convex, then $P \widehat{\nabla} P'$ is defined by the constraints in P which are also satisfied by P'. For example,

$$(x - 1 \leq y \leq x) \ \widehat{\nabla}\ (x - 2 \leq y \leq x) \equiv y \leq x$$

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 $P \widehat{\nabla} P'$ is built by simply removing constraints from P and since we cannot remove infinitely many constraints, the finiteness property is satisfied.

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

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 $\forall\Box(y \neq a)$, where a is a positive constant. Our symbolic model checker will convert this property to $\neg(\mathsf{True}\ \exists\mathcal{U}\ (y = a))$, and first try to compute an exact fixpoint for $\mathsf{True}\ \exists\mathcal{U}\ (y = a)$. Figure 8 shows the regions Q_0, Q_1, Q_2 generated by the first iterations of the exact algorithm. At this point, we can see that the sequence will diverge.

If we use a widening sequence with seed $s = 1$, then $\hat{Q}_0 \equiv Q_0$ and $\hat{Q}_1 \equiv Q_1$, and then note that $Q_2 \equiv \hat{Q}_1 \vee (\mathsf{True}\ \wedge\ \mathbf{pre}(\hat{Q}_1))$. We then obtain $\hat{Q}_2$ by computing $Q_1 \widehat{\nabla} Q_2$:

$$\begin{aligned} Q_1 \widehat{\nabla} Q_2 &\equiv (a \leq x + y\ \wedge\ a - 1 \leq y \leq a)\ \widehat{\nabla}\ (a \leq x + y \wedge a - 2 \leq y \leq a) \\ &\equiv (a \leq x + y \wedge y \leq a) \end{aligned}$$

The iterations converge, since this formula is also generated for $\hat{Q}_3$. When we negate the result, we get $x + y < a \vee a < y$. In other words, if our initialization of x and y satisfies this condition, then the invariant will indeed hold.

There are several points that should be clarified. First, note that the widening operator works on the syntax of the formula. Hence, different representations of the same formula may give different results when fed into the widening operator. In the simple example discussed above, if we start with $s = 0$, we end up computing $\hat{Q}_1$ via $Q_0 \widehat{\nabla} Q_1$, where

$$Q_0 \equiv y = a \qquad \text{and} \qquad Q_1 \equiv a \leq x + y\ \wedge\ a - 1 \leq y \leq a.$$

Then the result will be $Q_0 \widehat{\nabla} Q_1 \equiv \mathsf{True}$. And indeed, in our experiments, we found that most widening sequences require higher seeds to get started. By using a higher seed, we gain more *exact* information about the program before taking any approximations – and hence, loop growth directions can be (roughly) predicted.

Another way to prevent over approximations is to avoid using the widening operator when two polytopes have different dimensions. Note that for the example given above Q_0 is a line (i.e., a one-dimensional polytope) whereas Q_1 is a plane (i.e., a two-dimensional polytope).

In our implementation, we start with a low seed (between $s = 0$ and $s = 2$). If the approximation is too coarse, we gradually ratchet up seed's value, and generate a new widening sequence. In this way, successively tighter approximations can be obtained. We also allow selective bounding of the seed – since, after all, there may be cases where a property cannot be proved or disproved.

The widening operator $\widehat{\nabla}$ is sufficient if we always have convex sets. 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. Since the widening operator $\widehat{\nabla}$ folds all arguments into a single convex region, a direct application of this method failed to work. 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 $\widehat{\nabla}$ 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:

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

where all the q_i's and r_i's are convex polytopes. In Figure 9 we present how our multi-polyhedra widening operator is computed with such an input. The until loop (lines 1 to 9) reduces the number

ALGORITHM FOR MULTI-POLYHEDRA WIDENING

```
INPUT $Q \equiv \bigvee_{j=1}^{m} q_j$, $R \equiv \bigvee_{i=1}^{n} r_i$  s.t.  $q_j, r_i$  are all convex polytopes, and  $Q \subseteq R$
OUTPUT $P \equiv Q \bigtriangledown R$

1   REPEAT
2      $R' \leftarrow$ False; $marked[1..n] \leftarrow$ False; $deleted \leftarrow 0$;
3      FOR  $1 \leq i < j \leq n$  DO
4         IF  $marked[i] \equiv marked[j] \equiv$ False $\wedge$ $\mathbf{hull}(r_i, r_j) \equiv r_i \vee r_j$
5         THEN  $R' \leftarrow R' \vee \mathbf{hull}(r_i, r_j)$; $marked[i], marked[j] \leftarrow$ True; $deleted \leftarrow deleted + 1$;
6      FOR  $1 \leq i \leq n$  DO
7         IF  $marked[i] \equiv$ False  THEN  $R' \leftarrow R' \vee r_i$;
8      $R \leftarrow R'$; $n \leftarrow n - deleted$;
9   UNTIL $deleted \equiv 0$
10  $P \leftarrow$ False; $marked[1..n] \leftarrow$ False;
11  FOR  $1 \leq i \leq n \ \wedge\ 1 \leq j \leq m$  DO
12     IF  $q_j \subseteq r_i$  THEN  $P \leftarrow P \ \vee\ q_j \widehat{\bigtriangledown} r_i$ ; $marked[i] \leftarrow$ True;
13  FOR  $1 \leq i \leq n$  DO
14     IF  $marked[i] \equiv$ False  THEN  $P \leftarrow P \vee r_i$;
15  RETURN P
```

Figure 9: Multi-Polyhedra Widening.

of disjuncts in R by merging adjacent convex polytopes. Two convex polytopes are replaced by their hull if their hull is equal to their union. We continue doing this until no reduction is possible. Note that the resulting representation of R depends on the order of execution of the first for loop (lines 3 to 5). Traversing the list of polytopes in different order may lead to different results.

After minimizing R, we look for convex polytopes (i.e., disjuncts) in Q and R such that $q_j \subseteq r_i$. When we find such a pair, we use Cousot and Halbwachs' widening operator to compute $q_j \widehat{\bigtriangledown} r_i$. This new convex polytope is appended to the output Presburger set P. Finally we copy the disjuncts from R which are not widened to P, and return P as the result. Note that the order of the for loop in lines 11 to 12 can again effect the result.

The final result may include too many disjuncts. 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). In the experiments we conducted so far we never had to resort to this technique.

6.3 Computing Lower Bounds

Recall that each iteration of an exact fixpoint computation will yield a lower a bound for $f_1 \ \exists\mathcal{U} \ f_2$ and $f_1 \ \forall\mathcal{U} \ f_2$. 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 Running Example. We analyzed the running example given in Figure 2 and its requirement $\forall\Box(p \geq c)$. Using the negation-labeling algorithm, the requirement is rendered as $(\neg(\exists\Diamond(p \geq c)^{+})^{+})^{-}$. The temporal operator $\exists\Diamond$ is marked with "+" which means that we need an upper bound for the set of states violating the requirement. The symbolic model checker computes the upper bound using the multi-polyhedra widening technique, and it converges after 2 iterations. The result is the set $\hat{Q}$ which is partitioned as

$$\hat{Q}_{Idle} \equiv pcs = Idle \ \wedge p < c + q_1 + q_2 \qquad\qquad \hat{Q}_{Send} \equiv pcs = Send \ \wedge p < c + q_1 + q_2$$

However, since we are actually computing $\neg\exists\Diamond(p \geq c)$, the model checker computes $\neg\hat{Q}$, which gives a lower approximation for the states which satisfy the property $\forall\Box(p \geq c)$. Recall that the set of initial states of the running example is $I \equiv p = 0 \wedge c = 0 \wedge q_1 = 0 \wedge q_2 = 0$, and observe that $I \subseteq \neg\hat{Q}$. Hence, the model checker reports that the property is proved (with a CPU time of 1.74 seconds).

7 Reachability Analysis

In the previous section we discussed state-based approximations for general model-checking decision problems, where we have to verify a general CTL formula over infinite state-spaces. However, we also make use of two special-purpose techniques, which fall into the generic category of reachability-analysis. One variant of this is *state-based*, and it computes an upper bound for the program's reachable state-space. The other method is *transition-based*, it produces approximations for the transitive closure of the transition system itself. Both of these techniques have proved most successful when used *in conjunction* with our symbolic model-checker – both for exact and conservative analysis.

7.1 State-Based Reachability Analysis

The fixpoint algorithms described thus far are *backward* techniques, in that they start with a property f, and then use **pre** 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 RS, and then use RS^+ to help in the model-checking process. We can accomplish practically this by altering the symbolic model checker to restrict its computations to states in RS^+.

To generate the upper bound RS^+, we define a function **post** : $2^S \rightarrow 2^S$, called the *postcondition function*, which, given a set of states, returns the states reachable from this set in one step; i.e., $\mathbf{post}(Q) \overset{\text{def}}{=} \{ s \mid \exists s' : s' \in Q \wedge (s', s) \in X\}$. In particular, note that this is the "forward analogue" to the **pre** function. We claim that the (exact) reachable state space of a program is the least fixpoint of the functional

$$\lambda y \,.\, I \vee \mathbf{post}(y)$$

and which can be computed using the techniques we previously developed for $\exists\mathcal{U}$ and $\forall\mathcal{U}$. Moreover, we can use the widening method to compute an upper bound for RS as well.

After computing RS^+, we restrict the result of every operation in the model checker (Figure 7) to RS^+. For example, when a fixpoint iterate Q_i is produced, it is replaced by $Q_i \wedge RS^+$. Most importantly, we also can use this technique when we compute approximate fixpoints, as defined above.

7.2 Transition-Based Reachability Analysis

We also developed a transition-based reachability analysis technique, in which multiple execution paths get collapsed into single relations. We achieve this by adding new events to the original program; these events summarize repeated executions of self-loops, while preserving the underlying state-space. These transformations make the fixpoint computations for reachability properties converge faster without changing their truth sets.

Given a program $C = (V, I, E)$, we define a set of programs $T(C)$ as follows:

$$T(C) = \{ \, (V, I, E') \mid \bigvee_{e \in E} X_e \subseteq \bigvee_{e \in E'} X_e \subseteq (\bigvee_{e \in E} X_e)^* \equiv X^* \}$$

where X^* denotes the transitive closure of the transition relation X.

The programs in $T(C)$ have the following property:

Property 2 *If $C' \in T(C)$, then (1) it includes at least all the single step executions in C, and (2) for every finite execution path in C' there exists a finite execution path in C such that the first and the last states of the two execution paths are the same.*

Using definition of $T(C)$ and Property 2 we can show the following:

Property 3 *If $C' \in T(C)$ then*

1. *Truth sets of state formulas $f \in SF$ for C and C' are identical.*
2. *The set of reachable states RS for C and C' are identical.*
3. *If the truth set of the formula f for C' and C are the same then the truth sets of formulas $\exists\Diamond f$ and $\forall\Box f$ are the same for C' and C.*
4. *If the truth sets of the formulas f and g for C' are supersets of the truth sets of the corresponding formulas for C, then the truth sets of formulas $f\ \exists\mathcal{U}\ g$ and $\exists \bigcirc f$ for C' give upper bounds for the truth sets of the corresponding formulas for C.*
5. *If truth sets of the formulas f and g for C' are subsets of the truth sets of the corresponding formulas for C, then the truth sets of formulas $f\ \forall\mathcal{U}\ g$ and $\forall \bigcirc f$ for C' give lower bounds for the truth sets of the corresponding formulas for C.*

Theoretically, if we could compute X^* exactly, we would have a single relation which would summarize all reachable executions of the original program. That is, X^* would take any initial program state s as input, and instantly produce all states reachable from s. Likewise, $(X^*)^{-1}$ could handle any reverse reachable-state query. In other words, we could compute $\exists\Diamond f$ and RS in a single step. In general, though, constructing transitive closure of a Presburger relation is not computable.

However, akin to our state-based techniques, there are approximation techniques for obtaining upper and lower bounds for X^* [22, 23]. In turn, these can be used to compute our desired lower and upper bounds for RS and $\exists\Diamond f$. However, there's a fundamental trade-off involved in this: On one hand, if X is a simple convex relation in a few dimensions, it may be easy to compute X^* exactly, or a reasonable approximation thereof. Yet for such simple programs, we rarely need automated analysis. On the other hand, if a transition relation X is relatively complex (i.e., with many variables and concavities), the only tractable upper bound for X^* may be a trivial one – e.g., $S \times S$. Similarly, the computed lower bound may just turn out to be the identity relation.

Hence, we use these approximation techniques on selected parts of X – and never on the whole program at once. Specifically, we compute transitive closures on selected self loops, which are stable with respect to our partition classes.

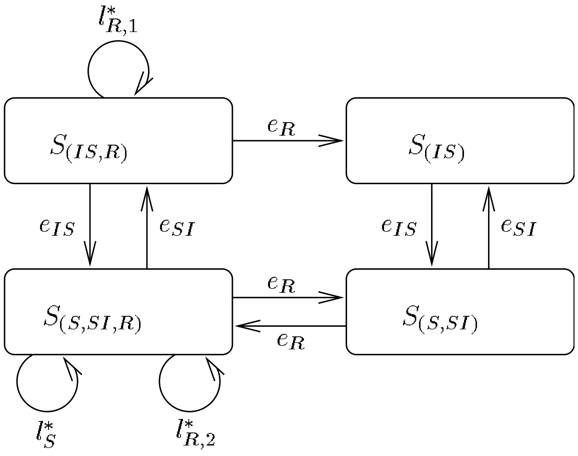

Figure 10: Running example with event-domain partitions and loop closures.

We apply this strategy automatically as follows. Given a program $C = (V, I, E)$, with a state space partitioning $P = \{S_1, S_2, \ldots S_p\}$, we compute the maximal set of self loops, where each loop $l_{e,i}$ is nonempty, and satisfies the following property:

$$l_{e,i} = \{ \ (s, s') \mid (s, s') \in X_e \wedge s \in S_i \wedge s' \in S_i\}$$

In other words, the loop $l_{e,i}$ represents stability – when it starts on a state in partition class S_i, it returns to the partition class S_i.

Next, for all such loops, we attempt to compute their transitive closures $l^*_{e,i}$. If obtaining the exact $l^*_{e,i}$ proves impossible, our analyzer gets a lower-bound approximation. (Note that this is often sufficient.) Then we generate a new program $C' = (V, I, E')$, by adding all events $l^*_{e,i}$ (or their lower bounds) to E, and deleting redundant transitions. Since

$$\bigvee_{e\in E} X^*_e \subseteq (\bigvee_{e\in E} X_e)^* \quad \text{and} \quad \bigvee_{e\in E'} X_e \subseteq \bigvee_{e\in E} X^*_e \quad \text{we have} \quad \bigvee_{e\in E} X_e \subseteq \bigvee_{e\in E'} X_e \subseteq (\bigvee_{e\in E} X_e)^*.$$

Therefore, $C' \in T(C)$ and satisfies Property 2 and Property 3

If we apply this technique to our unbounded buffer example from Figure 2, along with the event-domain partitioning, we get the program shown in Figure 10 where the three new events $l^*_{R,1}$, $l^*_{R,2}$ and l^*_S are self loop closures which are computed as explained above.

$$\begin{aligned}
l^*_{R,1} &\equiv (pc = \mathit{Idle} \wedge pc' = \mathit{Idle}) \wedge (c' = q_1 + q_2 + c - q'_1 - q'_2) \wedge (0 \le q'_1 \le q_1) \wedge (0 \le q'_2 \le q_2) \\
&\quad \wedge ((1 \le q'_1) \vee (1 \le q'_2)) \\
l^*_{R,2} &\equiv (pc = \mathit{Send} \wedge pc' = \mathit{Send}) \wedge (c' = q_1 + q_2 + c - q'_1 - q'_2) \wedge (0 \le q'_1 \le q_1) \wedge (0 \le q'_2 \le q_2) \\
&\quad \wedge ((1 \le q'_1) \vee (1 \le q'_2)) \\
l^*_S &\equiv (pc = \mathit{Send} \wedge pc' = \mathit{Send}) \wedge (p' = p - q_1 - q_2 + q'_1 + q'_2) \wedge (0 \le q_1 \le q'_1) \wedge (0 \le q_2 \le q'_2) \\
&\quad \wedge ((1 \le q_1) \vee (1 \le q_2))
\end{aligned}$$

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:**	$pc_2' = T_2 \wedge b' = 0$

Figure 11: The bakery algorithm.

8 Example Concurrent Programs

In this section we will present several example concurrent programs and discuss how we analyzed them using our symbolic model checker.

8.1 Bakery Algorithm

Consider the concurrent program shown in Figure 11, which implements the *bakery algorithm* [3] 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.

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).

The bakery algorithm is known to preserve both *mutual exclusion* and *starvation-freedom.* Based on our temporal logic, the bakery algorithm's mutual-exclusion property can be 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.* As for starvation-freedom, the property of interest is as follows: *if a process starts waiting for entry to the critical section, it eventually gets in.* For the first process, this can be expressed as: $\forall\Box(pc_1 = W_1 \rightarrow \forall\Diamond(pc_1 = C_1))$.

We analyzed the bakery algorithm using control point partitioning and the exact model checking procedure given in Figure 7. The mutual exclusion requirement for the bakery algorithm 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 $Q_0 = (pc_1 = C_1 \wedge pc_2 = C_2)$. After 4 iterations, the fixpoint computation converged to a set Q, where Q is partitioned as follows:

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:**	$pc_2' = T_2 \wedge s' = s + 1$

Figure 12: The ticket mutual-exclusion algorithm.

$Q_{(T_1,T_2)} \;:\; pc_1 = T_1 \wedge pc_2 = T_2 \wedge \mathsf{False}$

$Q_{(T_1,W_2)} \;:\; pc_1 = T_1 \wedge pc_2 = W_2 \wedge b = 0$

$Q_{(T_1,C_2)} \;:\; pc_1 = T_1 \wedge pc_2 = C_2 \wedge b = 0$

$Q_{(W_1,T_2)} \;:\; pc_1 = W_1 \wedge pc_2 = T_2 \wedge a = 0$

$Q_{(W_1,W_2)} \;:\; pc_1 = W_1 \wedge pc_2 = W_2 \wedge (a = b = 0 \vee a = 0 \wedge 1 \leq b \vee b = 0 \wedge 1 \leq a)$

$Q_{(W_1,C_2)} \;:\; pc_1 = W_1 \wedge pc_2 = C_2 \wedge (b = 0 \vee a < b)$

$Q_{(C_1,T_2)} \;:\; pc_1 = C_1 \wedge pc_2 = T_2 \wedge a = 0$

$Q_{(C_1,W_2)} \;:\; pc_1 = C_1 \wedge pc_2 = W_2 \wedge (a = 0 \vee b < a)$

$Q_{(C_1,C_2)} \;:\; pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{True}$

Then to obtain $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, the model checker computes $\neg Q$. This yields the set of states which can never reach a violation of the mutual exclusion property. The set of initial states for the bakery algorithm is $I \equiv pc_1 = T_1 \wedge pc_2 = T_2 \wedge a = b = 0$, and we see that that $I \subseteq \neg Q$ (i.e., all of the initial states satisfy the safety property). Hence, the model checker reports that the property is proved (for a total computation time of 2.85 seconds on a Sun SPARCstation 5).

The model checker in Figure 7 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).

8.2 Ticket Algorithm

In Figure 12, we present the *ticket algorithm* [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.

Again, the mutual-exclusion property is $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, which asserts that two processes can not be in the critical section at the same time. We first tried to check the mutual exclusion property of the ticket algorithm with the exact analyzer using control point partitioning. To compute the exact fixpoint, the analyzer started symbolically enumerating ways that both a and

b could be less than s. Since s and t are unbounded, this computation does not terminate. Next, we applied our conservative approximation technique. 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 symbolic model checker computes the upper bound using the multi-polyhedra widening technique, and it converges after 9 iterations. The result is the set $\hat{Q}$ which is partitioned as

$$\boxed{\begin{array}{lcl lcl}
\hat{Q}_{(T_1,T_2)} & : & pc_1 = T_1 \wedge pc_2 = T_2 \wedge t < s & \hat{Q}_{(W_1,T_2)} & : & pc_1 = W_1 \wedge pc_2 = T_2 \wedge t \leq s \\
\hat{Q}_{(T_1,W_2)} & : & pc_1 = T_1 \wedge pc_2 = W_2 \wedge t \leq s & \hat{Q}_{(C_1,T_2)} & : & pc_1 = C_1 \wedge pc_2 = T_2 \wedge t \leq s \\
\hat{Q}_{(T_1,C_2)} & : & pc_1 = T_1 \wedge pc_2 = C_2 \wedge t \leq s & \hat{Q}_{(C_1,C_2)} & : & pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{True} \\
\hat{Q}_{(W_1,C_2)} & : & \multicolumn{4}{l}{pc_1 = W_1 \wedge pc_2 = C_2 \wedge (a \leq s \vee t \leq s+1)} \\
\hat{Q}_{(C_1,W_2)} & : & \multicolumn{4}{l}{pc_1 = C_1 \wedge pc_2 = W_2 \wedge (b \leq s \vee t \leq s+1)} \\
\hat{Q}_{(W_1,W_2)} & : & \multicolumn{4}{l}{pc_1 = W_1 \wedge pc_2 = W_2 \wedge (b \leq s \wedge a \leq s \vee t \leq s+1 \wedge b \leq s \vee t \leq s+1 \wedge a \leq s)}
\end{array}}$$

However, since we are actually computing $\neg\exists\Diamond(pc_1 = C_1 \wedge pc_2 = C_2)$, the model checker computes $\neg\hat{Q}$, which gives a lower approximation for the states which respect mutual exclusion. Recall that the set of initial states of the ticket algorithm is $I \equiv pc_1 = T_1 \wedge pc_2 = T_2 \wedge t = s$, and observe that $I \subseteq \neg\hat{Q}$. Hence, the model checker reports that the property is proved (with a CPU time of 7.32 seconds).

We also wish to prove starvation-freedom. Negation-labeling converts process 1's relevant formula to:

$$(\neg(\exists\Diamond((pc_1 = W_1)^+ \wedge (\neg(\forall\Diamond(pc_1 = C_1)^-)^-)^+)^+)^+)^-.$$

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 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. 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 (for a total CPU time of 27.03 seconds).

We also tried to verify the ticket algorithm using reachability analysis. The analyzer computed the following upper bound for the set of reachable states RS^+ of the ticket algorithm (which turns out to be exact):

Program: Producer-Consumer
Data Variables: a, p_1, p_2, c_1, c_2: positive integer
Constants: s: integer, $s \geq 1$
Initial Condition: $p_1 = p_2 = c_1 = c_2 = 0 \wedge a = s$
Events:

e_{P_1}	**enabled:**	$a > 0$	e_{C_1}	**enabled:**	$a < s$
	action:	$p_1' = p_1 + 1 \wedge a' = a - 1$		**action:**	$c_1' = c_1 + 1 \wedge a' = a + 1$
e_{P_2}	**enabled:**	$a > 0$	e_{C_2}	**enabled:**	$a < s$
	action:	$p_2' = p_2 + 1 \wedge a' = a - 1$		**action:**	$c_2' = c_2 + 1 \wedge a' = a + 1$

Figure 13: A bounded-buffer producer-consumer program.

$RS^{+}_{(T_1,T_2)} \; : \; pc_1 = T_1 \wedge pc_2 = T_2 \wedge t = s$
$RS^{+}_{(T_1,W_2)} \; : \; pc_1 = T_1 \wedge pc_2 = W_2 \wedge t = s + 1 \wedge b = s$
$RS^{+}_{(T_1,C_2)} \; : \; pc_1 = T_1 \wedge pc_2 = C_2 \wedge t = s + 1 \wedge b = s$
$RS^{+}_{(W_1,T_2)} \; : \; pc_1 = W_1 \wedge pc_2 = T_2 \wedge t = s + 1 \wedge a = s$
$RS^{+}_{(C_1,T_2)} \; : \; pc_1 = C_1 \wedge pc_2 = T_2 \wedge t = s + 1 \wedge a = s$
$RS^{+}_{(C_1,C_2)} \; : \; pc_1 = C_1 \wedge pc_2 = C_2 \wedge \mathsf{False}$
$RS^{+}_{(W_1,C_2)} \; : \; pc_1 = W_1 \wedge pc_2 = C_2 \wedge t = s + 2 \wedge a = s + 1 \wedge b = s$
$RS^{+}_{(C_1,W_2)} \; : \; pc_1 = C_1 \wedge pc_2 = W_2 \wedge b = s + 1 \wedge a = s \wedge t = s + 2$
$RS^{+}_{(W_1,W_2)} \; : \; pc_1 = W_1 \wedge pc_2 = W_2 \wedge b + a = 2s + 1 \wedge t = 2 + s \wedge b - 1 \leq s \leq b$

When the state space of the ticket algorithm was restricted to this set the exact analyzer was able to prove both the mutual exclusion and the starvation freedom properties of the ticket algorithm.

8.3 Producer-Consumer

In Figure 13, we present a bounded-buffer producer-consumer problem adapted from an example in [30]. This program implements an instance of the problem with two producers and two consumers. We wish to prove that $\forall\Box(0 \leq p_1 + p_2 - (c_1 + c_2) \leq s)$ holds, i.e., that the bounded buffer properties are satisfied. When we translate this into existential form, we get $\neg\exists\Diamond(\neg(0 \leq p_1 + p_2 - (c_1 + c_2) \leq s))$. The exact model checker diverged when we fed it the producer-consumer program with this property. But when we tried the widening technique, the model checker successfully verified the property. We were also able to prove the same property using reachability analysis in conjunction with exact fixpoint computations.

8.4 Circular Queue Program

The circular queue program (Figure 14) consists of one producer component and one consumer component. The producer and consumer execute concurrently. Figure 15 shows some possible configurations of the queue during the execution of the program.

Program: Circular Queue
Data Variables: h, t, p, c: integer
Constants: s: integer, $s \geq 1$
Initial Condition: $h = t = p = c = 0$
Events:
e_P **enabled:** $(t = s \rightarrow h > 0) \wedge (t < s \rightarrow t+1 \neq h)$
action: $((t = s \wedge t' = 0) \vee (t < s \wedge t' = t+1)) \wedge p' = p+1$

e_C **enabled:** $(t \neq h)$
action: $((h = s \wedge h' = 0) \vee (h < s \wedge h' = h+1)) \wedge c' = c+1$

Figure 14: Circular queue program.

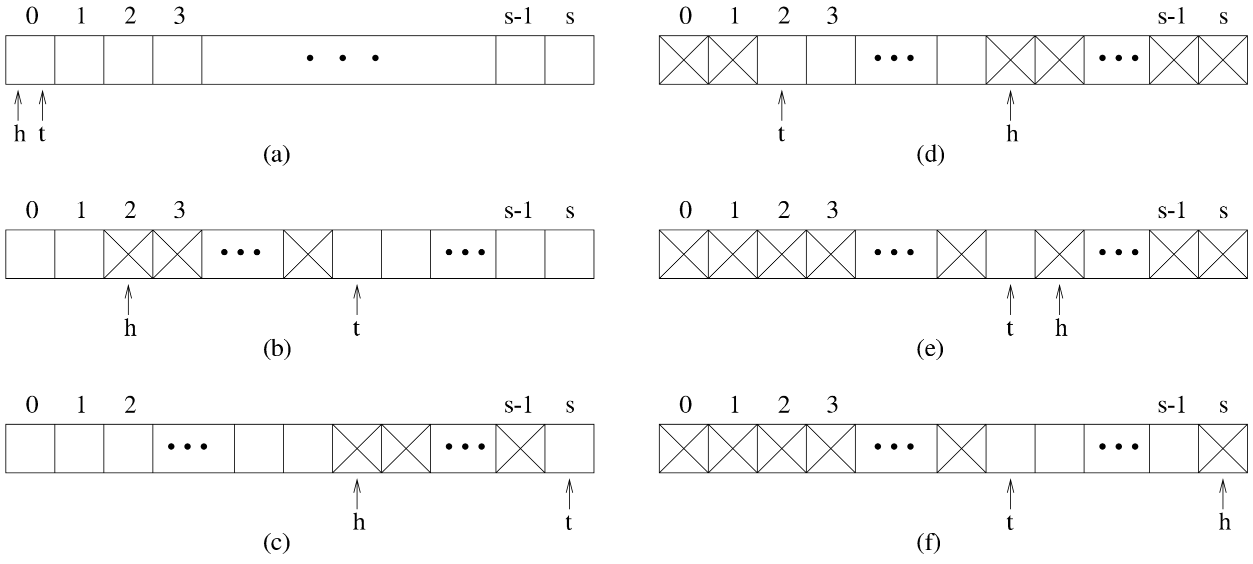

Figure 15: Possible Configurations of the Buffer During the Execution of the Circular Queue Program. (Crossed locations indicate occupied cells. Note that, (a) shows the initial configuration and (e) shows a configuration where the queue is full.)

Program: Circular Queue
Data Variables: h, t, p, c: integer
Constants: s: integer, $s \geq 1$
Initial Condition: $h = t = p = c = 0$
Events:

e_{P_1}	**enabled:**	$t < s \wedge h < t+1$	e_{C_1}	**enabled:**	$h < t$
	action:	$t' = t+1 \wedge p' = p+1$		**action:**	$h' = h+1 \wedge c' = c+1$
e_{P_2}	**enabled:**	$t < s \wedge h > t+1$	e_{C_2}	**enabled:**	$h < s \wedge h > t$
	action:	$t' = t+1 \wedge p' = p+1$		**action:**	$h' = h+1 \wedge c' = c+1$
e_{P_3}	**enabled:**	$t = s \wedge h > 0$	e_{C_3}	**enabled:**	$h = s \wedge h > t$
	action:	$t = 0 \wedge p' = p+1$		**action:**	$h' = 0 \wedge c' = c+1$

Figure 16: Circular queue program represented with DNF-Transformed events.

Variables h and t represent the *head* and the *tail* of the queue, respectively; h points to the location of the item that will be consumed next (if the queue is not empty), and t points to the location where the next produced item will be placed (if the queue is not full). Constant s denotes the size of the queue. Although there are $s+1$ locations in the queue, maximum number of items that can be stored is s. Variables p and c show the number of items produced and consumed, respectively.

There are several interesting properties we may want to prove about the circular queue program:

$$\forall\Box(h \leq s \wedge t \leq s), \qquad \forall\Box(0 \leq p - c \leq s),$$
$$\forall\Box(t \geq h \rightarrow p - c = t - h), \quad \forall\Box(t < h \rightarrow p - c = s - (h - t) + 1).$$

When we decompose the events of the circular queue program to DNF-decomposed events, we get a program with 6 events as shown in Figure 16. In this program the producer and the consumer components are represented by three events each. We observe that at any time only one producer event and one consumer event is enabled. The enabling conditions for events e_{P_1}, e_{P_2} and e_{P_3} are mutually exclusive (i.e., when one of them is enabled the rest are disabled). This is also true for the three consumer events. This means that there is some inherent sequential behavior in the producer and consumer components of the program. Based on this new event set, the event-domain partitioning algorithm generates a partitioning with 10 classes which is significantly less than the worst case, and for each partition class there are at most two events enabled.

After computing the event-domain partitioning based on DNF decomposition, we computed self-loop closures of the program. We first transform the program via a reachable-transition computation. After this transformation, the symbolic model checker in Figure 7 successfully verified all the safety properties of the circular queue on the transformed program. Hence, we can conclude

Program	Property	Label
Unbounded Buffer	$\forall\Box(p = c + q_1 + q_2)$	UB1
Unbounded Buffer	$\forall\Box((pc_S = \mathit{Idle} \wedge q_1 + q_2 = i) \rightarrow \forall\bigcirc(q_1 + q_2 \leq i))$	UB2
Unbounded Buffer	$\forall\Box(p \geq c)$	UB3
Bakery Algorithm	$\forall\Box(\neg(pc_1 = C_1 \wedge pc_2 = C_2))$	B1
Bakery Algorithm	$\forall\Box(pc_1 = W_1 \rightarrow \forall\Diamond(pc_1 = C_1))$	B2
Ticket Algorithm	$\forall\Box(\neg(pc_1 = C_1 \wedge pc_2 = C_2))$	T1
Ticket Algorithm	$\forall\Box(pc_1 = W_1 \rightarrow \forall\Diamond(pc_1 = C_1))$	T2
Producer Consumer	$\forall\Box(0 \leq p - (c_1 + c_2) \leq s)$	PC
Circular Queue	$\forall\Box(h \leq s \wedge t \leq s)$	CQ1
Circular Queue	$\forall\Box(t \geq h \rightarrow p - c = t - h)$	CQ2
Circular Queue	$\forall\Box(t < h \rightarrow p - c = s - (h - t) + 1)$	CQ3
Circular Queue	$\forall\Box(0 \leq p - c \leq s)$	CQ4

Figure 17: List of problem instances used in the experiments.

that the original program satisfied all of the safety properties.

A summary of our experiments is shown in Figures 17 and 18. We have used five programs and several temporal properties to test our analyzer (properties are listed in Figure 17). The results of the tests are shown in Figure 18.

9 Conclusions

We have presented a new symbolic model checker for infinite-state programs, which evaluates safety and liveness properties. Some features of our symbolic model checker are as follows: (1) it symbolically encodes transition relations and state sets of programs using Presburger formulas, which can be manipulated efficiently using the Omega library; (2) it partitions a program's state-space via the control variables or domains of events, and uses the partition classes as repositories for the model checker's symbolic computations; and (3) it approximates the uncomputable fixpoint computations with techniques that guarantee convergence in finite time. We demonstrated our method using five infinite-state concurrent programs, which exploited the following analysis techniques: exact-backward, exact-forward, approximate-backward, approximate-forward and reachability 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. Our current symbolic encoding treats every program variable as an integer. This is obviously not an efficient way to handle variables with small domains (e.g. boolean variables). We are currently working on integrating different symbolic representations in a single model checker so that every variable will be represented with a suitable symbolic represen-

Problem Instance	DNF Transition Decomposition	State Partitioning	Transition-Based Reachability	State-Based Reachability	Fixpoint Computations	Widening Seed	Execution Time
UB1					Exact		0.73 sec.
UB2					Exact		0.67 sec.
UB3					Exact		↑
UB3					Approximate	0	1.37 sec.
UB3			√		Exact	0	4.18 sec.
B1		Control Point			Exact		2.85 sec.
B1		Control Point			Approximate	0	3.21 sec.
B2		Control Point			Exact		7.64 sec.
B2		Control Point			Approximate	0	4.72 sec.
T1		Control Point			Exact		↑
T1		Control Point		√	Exact		1.44 sec.
T1		Control Point			Approximate	0	7.32 sec.
T2		Control Point			Exact		↑
T2		Control Point		√	Exact		4.34 sec.
T2		Control Point			Approximate	0	27.03 sec.
PC					Exact		↑
PC					Approximate	0	5.34 sec.
PC			√		Exact	0	4.25 sec.
PC		Event-Domain	√		Exact	0	37.47 sec.
CQ1					Exact		0.54 sec.
CQ1					Approximate	0	0.6 sec.
CQ1	√	Event-Domain	√		Exact		11.15 sec.
CQ2					Exact		↑
CQ2	√	Event-Domain			Exact		↑
CQ2	√	Event-Domain			Approximate	4	420.84 sec.
CQ2	√	Event-Domain	√		Exact		18.91 sec.
CQ3					Exact		↑
CQ3	√	Event-Domain			Approximate	0	37.06 sec.
CQ3	√	Event-Domain	√		Exact		25.53 sec.
CQ4					Exact		↑
CQ4	√	Event-Domain			Approximate	3	231.94 sec.
CQ4	√	Event-Domain		√	Exact		↑
CQ4	√	Event-Domain	√		Exact		119.44 sec.

Figure 18: Summary of the Experiments (↑ denotes that the system diverged).

tation. Hence, a set of states will be represented by a hybrid symbolic representation which may consist of, for example, a BDD formula and a Presburger formula. Given the richness of different symbolic representations developed recently (e.g. BDDs, QDDs, real time model checkers), we think that this is a very promising direction. But it is also a an ambitious goal because of the difficulty of manipulating and simplifying such a representation. We predict that even with some very simple manipulations we can get useful results and extend the scope of current model checkers.

We are also working on developing better state partitioning techniques. For many reasons, getting the right partitioning of the state space can be crucial when carrying out the analysis; for example, it can greatly affect the performance of the widening technique. Our current partitioning techniques depend on the source of the program. We are using the information expressed by the programmer to partition the state space. Another direction is to use fixpoint computations on the transition system to compute bisimulations. Since this is an undecidable problem, again, we have to use approximation techniques.

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] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering 22(3)*, March 1996.

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

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

[5] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] 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. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[7] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counter-examples. In *Proc. of ACM Sigsoft ISSTA'96*, pages 224–238, January 1996.

[8] 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).

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

[10] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the 4th Annual IEEE Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[11] 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.

[12] 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 Symposium on Principles of Programming Languages*, 1977.

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

[14] R. Cridlig. Semantic analysis of concurrent ML by abstract model-checking. In *Proceedings of INFINITY International workshop on Infinite State Systems*, pages 71-86, 1996, Technical Report MIP-9614, University of Passau.

[15] L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4):643–669, (October 1990).

[16] J. Dingel, and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. Proceedings of CAV'95, LNCS 939, 1995.

[17] M. J. Fischer and M. O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. *SIAM-AMS Proceedings*, Volume 7, pages 27-41, 1974.

[18] 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.

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

[20] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report 95-31, Department of Computer Science, Eindhoven University of Technology, 1995.

[21] 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>.

[22] W. Kelly, and W. Pugh. Using affine closure to find legal reordering transformations. *International Journal of Parallel Programming*, 1995.

[23] W. Kelly, W. Pugh, E. Rosser and T. Shpeisman. Transitive closure of infinite graphs and its applications. In *Eight Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995. Technical Report CS-TR-3457, UMIACS-TR-95-48, Department of Computer Science, University of Maryland, April 1994.

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

[25] K. L. McMillan. Symbolic model checking. Massachusetts, 1993, Kluwer Academic Publishers.

[26] 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).

[27] A. Pardo and G. D. Hachtel. Automatic Abstraction Techniques for Propositional μ-calculus Model Checking. Proceedings of CAV '97, LNCS 1254, Orna Grumberg, ed., pages 12–23, 1997.

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

[29] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems,* 14(3):1248-1278, July 1994.

[30] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, (September 1993).