Program Specialization for Verifying Infinite State Systems: An Experimental Evaluation

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[3], and Valerio Senni[2]

[1] Dipartimento di Scienze, University 'G. D'Annunzio', Pescara, Italy
fioravanti@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[3] CNR-IASI, Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We address the problem of the automated verification of temporal properties of infinite state reactive systems. We present some improvements of a verification method based on the specialization of constraint logic programs (CLP). First, we reformulate the verification method as a two-phase procedure: (1) in the first phase a CLP specification of an infinite state system is specialized with respect to the initial state of the system and the temporal property to be verified, and (2) in the second phase the specialized program is evaluated by using a bottom-up strategy. In this paper we propose some new strategies for performing program specialization during the first phase. We evaluate the effectiveness of these new strategies, as well as that of some old strategies, by presenting the results of experiments performed on several infinite state systems and temporal properties. Finally, we compare the implementation of our specialization-based verification method with various constraint-based model checking tools. The experimental results show that our method is effective and competitive with respect to the methods used in those other tools.

1 Introduction

One of the most challenging problems in the verification of reactive systems is the extension of the model checking technique (see [9] for a thorough overview) to infinite state systems. In model checking the evolution over time of an infinite state system is modelled as a binary transition relation over an infinite set of states and the properties of that evolution are specified by means of propositional temporal formulas. In particular, in this paper we consider the *Computation Tree Logic* (CTL), which is a branching time propositional temporal logic by which one can specify, among others, the so-called *safety* and *liveness* properties [9].

Unfortunately, the verification of CTL formulas for infinite state systems is, in general, an undecidable problem. In order to cope with this limitation, various *decidable subclasses* of systems and formulas have been identified (see, for instance, [1,15]). Other approaches enhance finite state model checking by using more general *deductive* techniques (see, for instance, [33,37]) or using *abstractions*, by which one can compute conservative approximations of the set of states verifying a given property (see, for instance, [2,6,8,11,19,20]).

Also logic programming and constraint logic programming (CLP) have been proposed as frameworks for specifying and verifying properties of reactive systems. Indeed, the fixpoint semantics of logic programming languages allows us to easily represent the fixpoint semantics of various temporal logics [14,30,34]. Moreover, constraints over the integers or the rationals can be used to provide finite representations of infinite sets of states [14,18].

However, for programs that specify infinite state systems, the proof procedures normally used in constraint logic programming, such as the extensions to CLP of SLDNF resolution and tabled resolution [7], very often diverge when trying to check some given temporal properties. This is due to the limited ability of these proof procedures to cope with infinitely failed derivations. For this reason, instead of simply applying program evaluation, many logic programming-based verification systems make use of reasoning techniques such as: (i) *abstract interpretation* [4,14] and (ii) *program transformation* [16,26,28,31,35].

In this paper we further develop the verification method presented in [16] and we assess its practical value. That method is applicable to specifications of CTL properties of infinite state systems encoded as constraint logic programs and it makes use of program specialization.

The specific contributions of this paper are the following. First, we have reformulated the specialization-based verification method of [16] as a two-phase method. In Phase (1) the CLP specification is specialized with respect to the initial state of the system and the temporal property to be verified, and in Phase (2) the construction of the perfect model of the specialized program is performed via a bottom-up evaluation. The main goal of Phase (1) is to derive a specialized program for which the bottom-up model construction of Phase (2) terminates. We have shown in an experimental way that, indeed, Phase (2) terminates in most examples without the need for abstractions.

We have defined various generalization strategies which can be used during Phase (1) of our verification method for controlling when and how to perform generalization. The selection of a good generalization strategy is not a trivial task: the selected strategy must guarantee the termination of the specialization phase and should also provide a good balance between precision (that is, the number of properties that are proved) and verification time. Indeed, the use of a too coarse generalization strategy may prevent the proof of the properties of interest, while an unnecessarily precise strategy may lead to verification times which are too high. Since the states of the systems we consider are encoded as n-tuples of rationals, our generalization strategies have been specifically designed for CLP programs using linear inequations over rationals as constraints.

We have implemented our strategies using the MAP transformation system [29] and we have compared them in terms of precision and efficiency on several infinite state systems taken from the literature. Finally, we have compared our MAP implementation with some constraint-based model checkers for infinite state systems and, in particular, with ALV [39], DMC [14], and HyTech [21].

The paper is structured as follows. In Section 2 we recall how CTL properties of infinite state systems can be encoded as locally stratified CLP programs. In Section 3 we present our two-phase verification method. In Section 4 we describe

various strategies that can be applied during the specialization phase and, in particular, the generalization strategies used for ensuring termination of that phase. In Section 5 we report on the experiments we have performed by using a prototype implemented on our MAP transformation system.

2 Specifying CTL Properties by CLP Programs

We will follow the approach presented in [9] and we will model an infinite state system as a *Kripke structure*. The properties to be verified will be specified as formulas of the *Computation Tree Logic* (CTL). The fact that a CTL formula φ holds in a state s of a Kripke structure $\mathcal{K}$ will be denoted by $\mathcal{K}, s \models \varphi$.

A Kripke structure $\langle S, I, R, L\rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, R is a transition relation, and L is a labeling function, can be encoded as a CLP program as follows. (1) A state in S is encoded as an n-tuple of the form $\langle t_1, \ldots, t_n\rangle$, where for $i = 1, \ldots, n$, the term t_i is either a rational number or an element of a finite domain. For reasons of simplicity, when denoting a state we will feel free to use a single variable X, instead of an n-tuple of variables of the form $\langle X_1, \ldots, X_n\rangle$.

(2) An initial state X in I is encoded as a clause of the form:
$initial(X) \leftarrow c(X)$, where $c(X)$ is a constraint.

(3) The transition relation R is encoded as a set of clauses of the form:
$t(X, Y) \leftarrow c(X, Y)$
where $c(X, Y)$ is a constraint. The state Y is called a *successor state* of X. We also introduce a predicate ts such that, for every state X, $ts(X, Ys)$ holds iff Ys is a list of all the successor states of X, that is, for every state X, the state Y belongs to the list Ys iff $t(X, Y)$ holds. In [17] the reader will find: (i) an algorithm for deriving the clauses defining ts from the clauses defining t, and also (ii) conditions that guarantee that Ys is a finite list.

(4) The elementary properties which are associated with each state X by the labeling function L, are encoded as a set of clauses of the form:
$elem(X, e) \leftarrow c(X)$
where e is an elementary property and $c(X)$ is a constraint.

The satisfaction relation $\models$ can be encoded by a predicate sat defined by the following clauses [16] (see also [28,30] for similar encodings):

1. $sat(X, F) \leftarrow elem(X, F)$
2. $sat(X, not(F)) \leftarrow \neg sat(X, F)$
3. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1), sat(X, F_2)$
4. $sat(X, ex(F)) \leftarrow t(X, Y), sat(Y, F)$
5. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_2)$
6. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_1), t(X, Y), sat(Y, eu(F_1, F_2))$
7. $sat(X, af(F)) \leftarrow sat(X, F)$
8. $sat(X, af(F)) \leftarrow ts(X, Ys), sat\_all(Ys, af(F))$
9. $sat\_all([\,], F) \leftarrow$
10. $sat\_all([X|Xs], F) \leftarrow sat(X, F), sat\_all(Xs, F)$

We assume the perfect model semantics for our CLP programs.

Note that all the CTL operators considered in [9] can be defined in terms of ex, eu, and af. In particular, for every CTL formula φ, $ef(\varphi)$ can be defined as $eu(true, \varphi)$ and $eg(\varphi)$ can be defined as $not(af(not(\varphi)))$. By restricting the operators to ex, eu, and af, we are able to provide the straightforward encoding of the CTL satisfaction relation as the constraint logic program shown above. Note, however, that by using this subset of operators, we cannot rewrite all formulas in *negation normal form* (where negation appears in front of elementary properties only), which is sometimes used in model checking [9]. Dealing with formulas in negation normal form avoids the use of a non-monotonic immediate consequence operator, but it requires the construction of both the least and the greatest fixpoint of that operator. The use of greatest fixpoints would force us to prove the correctness of the program transformation rules we use for program specialization, while, if we use least fixpoints only (which are the only fixpoints required for defining the perfect model semantics) the correctness of those rules is an immediate consequence of the results in [36].

Given a CTL formula φ we define a predicate *prop* as follows:

$prop \equiv_{def} \forall X(initial(X) \rightarrow sat(X, \varphi))$

This definition can be encoded by the following two clauses:

$\gamma_1:\ prop \leftarrow \neg negprop$

$\gamma_2:\ negprop \leftarrow initial(X), sat(X, not(\varphi))$

The correctness of the encoding of CTL is stated by the following Theorem 1 (whose proof is given in [17]), where $P_{\mathcal{K}}$ denotes the constraint logic program consisting of clauses 1–10 together with the clauses defining the predicates *initial*, *t*, *ts*, and *elem*. Note that program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is locally stratified and, hence, it has a unique perfect model which will be denoted by $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ [3].

Theorem 1 (Correctness of Encoding). *Let $\mathcal{K}$ be a Kripke structure, let I be the set of initial states of $\mathcal{K}$, and let φ be a CTL formula. Then,*

for all states $s \in I$, $\mathcal{K}, s \models \varphi$ iff $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$.

Example 1. Let us consider the reactive system depicted in Figure 1, where a state $\langle X_1, X_2 \rangle$ which is a pair of rationals, is denoted by the term $s(X_1, X_2)$. In any initial state of this system we have that $X_1 \leq 0$ and $X_2 = 0$. There are two transitions: one from state $s(X_1, X_2)$ to state $s(X_1, X_2 - 1)$ if $X_1 \geq 1$, and one from state $s(X_1, X_2)$ to state $s(X_1, X_2 + 1)$ if $X_1 \leq 2$.

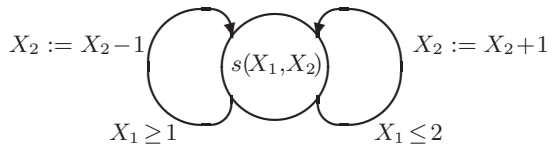

Fig. 1. A reactive system. The transitions do not change the value of X_1.

The Kripke structure $\mathcal{K}$ which models that system is defined as follows. The initial states are given by the clause:

11. $initial(s(X_1, X_2)) \leftarrow X_1 \leq 0,\ X_2 = 0$

The transition relation R is given by the clauses:

12. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \geq 1,\ Y_1 = X_1,\ Y_2 = X_2 - 1$

13. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \leq 2,\ Y_1 = X_1,\ Y_2 = X_2 + 1$

The elementary property *negative* is given by the clause:

14. $elem(s(X_1, X_2), negative) \leftarrow X_2 < 0$

Let $P_{\mathcal{K}}$ denote the program consisting of clauses 1–14. We omit the clauses defining the predicate *ts*, which are not needed in this example.

Suppose that we want to verify that in every initial state $s(X_1, X_2)$, where $X_1 \leq 0$ and $X_2 = 0$, the CTL formula $not(eu(true, negative))$ holds, that is, from any initial state it cannot be reached a state $s(X_1', X_2')$ where $X_2' < 0$. By using the fact that $not(not(\varphi))$ is equivalent to φ, this property is encoded as follows:

γ_1: $prop \leftarrow \neg negprop$

γ_2: $negprop \leftarrow initial(X),\ sat(X, eu(true, negative))$ □

3 Verifying Infinite State Systems by Specializing CLP Programs

In this section we present a method for checking whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is a CLP specification of an infinite state system and *prop* is a predicate encoding the satisfiability of a given CTL formula.

As already mentioned, the proof procedures normally used in constraint logic programming, such as the extensions to CLP of SLDNF resolution and tabled resolution, very often diverge when trying to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ by evaluating the query *prop*. This is due to the limited ability of these proof procedures to cope with infinite failure.

Also the bottom-up construction of the perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ often diverges, because it does not take into account the information about the query *prop* to be evaluated, the initial states of the system, and the formula to be verified. Indeed, by a naive bottom-up evaluation, the clauses of $P_{\mathcal{K}}$ may generate infinitely many atoms of the form $sat(s, \psi)$. For instance, given a state s_0, an elementary property f that holds in s_0, and an infinite sequence $\{s_i \mid i \in \mathbb{N}\}$ of distinct states such that, for every $i \in \mathbb{N}$, $t(s_{i+1}, s_i)$ holds, clauses 5 and 6 generate by bottom-up evaluation the infinitely many atoms of the form: (i) $sat(s_0, f), sat(s_0, eu(true, f)), sat(s_0, eu(true, eu(true, f))), \ldots$, and of the form: (ii) $sat(s_i, eu(true, f))$, for every $i \in \mathbb{N}$.

In this paper we will show that the termination of the bottom-up construction of the perfect model can be improved by a prior application of program specialization. In particular, in this section we will present a verification algorithm which is a reformulation of the method proposed in [16] and consists of two phases: Phase (1), in which we specialize the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ with respect to the query *prop*, thereby deriving a new program $P_{\mathcal{S}}$ whose perfect model $M_{\mathcal{S}}$ satisfies the following equivalence: $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_{\mathcal{S}}$, and Phase (2), in which we construct $M_{\mathcal{S}}$ by a bottom-up evaluation.

The specialization phase modifies the $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ by incorporating into the specialized program $P_{\mathcal{S}}$ the information about the initial states and the formula to be verified. The bottom-up evaluation of $P_{\mathcal{S}}$ may terminate more often than the bottom-up evaluation of $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ because: (i) it generates only specialized atoms corresponding to the subformulas of the formula to be

verified, and (ii) it avoids the generation of an infinite set of $sat(s, \psi)$ atoms where the state s in unreachable from the initial states.

The Verification Algorithm
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$. *Output*: The perfect model $M_{\mathcal{S}}$ of a CLP program $P_{\mathcal{S}}$ such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_{\mathcal{S}}$.
(Phase 1) $Specialize(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}, P_{\mathcal{S}})$;
(Phase 2) $BottomUp(P_{\mathcal{S}}, M_{\mathcal{S}})$

The *Specialize* procedure of Phase (1) makes use of the following transformation rules only: definition introduction, positive unfolding, constrained atomic folding, removal of clauses with unsatisfiable body, and removal of subsumed clauses. Thus, Phase (1) is simpler than the specialization technique presented in [16] which uses also some extra rules such as negative unfolding, removal of useless clauses, and contextual constraint replacement.

The Procedure *Specialize*
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$. *Output*: A stratified program $P_{\mathcal{S}}$ such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M(P_{\mathcal{S}})$.

$P_{\mathcal{S}} := \{\gamma_1\}$; $\quad InDefs := \{\gamma_2\}$; $\quad Defs := \emptyset$;
while there exists a clause γ in $InDefs$
do $Unfold(\gamma, \Gamma)$;
$\quad Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$;
$\quad P_{\mathcal{S}} := P_{\mathcal{S}} \cup \Phi$;
$\quad InDefs := (InDefs - \{\gamma\}) \cup NewDefs$; $\quad Defs := Defs \cup NewDefs$;
end-while

The *Unfold* procedure takes as input a clause $\gamma \in InDefs$ of the form $H \leftarrow c(X), sat(X, \psi)$, where ψ is a ground term denoting a CTL formula, and returns as output a set Γ of clauses derived from γ as follows. The *Unfold* procedure first unfolds once γ w.r.t. $sat(X, \psi)$ and then applies zero or more times the unfolding rule as long as in the body of a clause derived from γ there is an atom of one of the following forms: (i) $initial(s)$, (ii) $t(s_1, s_2)$, (iii) $ts(s, ss)$, (iv) $elem(s, e)$, (v) $sat(s, e)$, where e is a constant, (vi) $sat(s, not(\psi_1))$, (vii) $sat(s, and(\psi_1, \psi_2))$, (viii) $sat(s, ex(\psi_1))$, and (ix) $sat\_all(ss, \psi_1)$, where ss is a non-variable list. Then the set of clauses derived from γ by applying the unfolding rule is simplified by removing: (i) every clause whose body contains an unsatisfiable constraint, and (ii) every clause which is subsumed by a clause of the form $H \leftarrow c$, where c is a constraint. Due to the structure of the clauses in $P_{\mathcal{K}}$, the *Unfold* procedure terminates for any $\gamma \in InDefs$.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and the set *Defs* of clauses, called *definitions*. A definition in *Defs* is a clause of the form $newp(X) \leftarrow d(X), sat(X, \psi)$ which can be used for folding. The *Generalize&Fold* procedure introduces a set *NewDefs* of new definitions (which are then added to *Defs*) and, by folding the clauses in Γ using the definitions in $Defs \cup NewDefs$, derives a new set Φ of clauses

which are added to the program P_S. An uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions and, therefore, it may make the *Specialize* procedure not to terminate. In order to guarantee termination, we will extend to constraint logic programs some techniques which have been proposed for controlling generalization in *positive supercompilation* [38] and *partial deduction* [24,27]. More details on the *Generalize&Fold* procedure will be given in the next section.

The output program P_S of the *Specialize* procedure is a *stratified* program and the procedure *BottomUp* computes the perfect model M_S of P_S by considering a stratum at a time, starting from the lowest stratum and going up to the highest stratum of P_S (see, for instance, [3]). Obviously, the model M_S may be infinite and the *BottomUp* procedure may not terminate.

In order to get a terminating procedure, we could compute an approximation of M_S by applying abstract interpretation techniques [10]. Indeed, in order to prove that $prop \in M_S$, we could construct a set $A \subseteq M_S$ such that $prop \in A$. Several abstract interpretation techniques have been proposed for definite CLP programs (see [22] for a tool that implements many such techniques based on polyhedra). However, integrating approximation mechanisms with the bottom-up construction of the perfect model, requires us to define suitable extensions of those techniques which compute both over-approximations and under-approximations of models, because of the presence of negation. In this paper we will not address the issue of defining those extensions and we will focus our attention on the design of the *Specialize* procedure only. In Section 5 we show that after the application of our *Specialize* procedure, the construction of the model M_S terminates in many significant cases.

Example 2. Let us consider the reactive system $\mathcal{K}$ of Example 1. We want to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$. Now we have that: (i) by using a traditional Prolog system, the evaluation of the query *prop* does not terminate in the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ because *negprop* has an infinitely failed SLD tree, (ii) by using the XSB tabled logic programming system, the query *prop* does not terminate because infinitely many *sat* atoms are tabled, and (iii) the bottom-up construction of $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ does not terminate because of the presence of clauses 5 and 6 as we have indicated at the beginning of this section.

By applying the *Specialize* procedure to the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ (with a suitable generalization strategy, as illustrated in the next section), we derive the following specialized program P_S:

γ_1. $prop \leftarrow \neg negprop$
γ_2'. $negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ new1(X_1, X_2)$
γ_3. $new1(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 = 0,\ Y_1 = X_1,\ Y_2 = 1,\ new2(Y_1, Y_2)$
γ_4. $new2(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 \geq 0,\ Y_1 = X_1,\ Y_2 = X_2 + 1,\ new2(Y_1, Y_2)$

Note that the *Specialize* procedure has propagated through the program P_S the constraint $X_1 \leq 0,\ X_2 = 0$ characterizing the initial states (see clause 11 of Example 1). This constraint, in fact, occurs in clause γ_3 and its generalization $X_1 \leq 0,\ X_2 \geq 0$ occurs in clause γ_4. The *BottomUp* procedure computes the perfect model of P_S, which is $M_S = \{prop\}$, in a finite number of steps. Thus, $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$. □

Most model checkers provide *witnesses* of existential formulas, when these formulas hold, and *counterexamples* of universal formulas, when these formulas do not hold [9]. Our encoding of the Kripke structure can easily be extended to provide witnesses of formulas of the form $eu(\varphi_1, \varphi_2)$ and counterexamples of formulas of the form $af(\varphi)$ by adding to the predicate *sat* an extra argument that recalls the sequence of states (or transitions) constructed during the verification of a given formula. For details, the reader may refer to [17].

4 Generalization Strategies

The design of a powerful generalization strategy should meet the following two conflicting requirements: (i) the introduction of new definitions that are as general as possible to guarantee the termination of the *Specialize* procedure, and (ii) the introduction of new definitions that are not too general to guarantee the termination of the *BottomUp* procedure. In this section we present several generalization strategies for coping with those conflicting requirements.

These strategies combine various by now standard techniques used in the fields of program transformation and static analysis, such as *well-quasi orderings*, *widening*, and *convex hull* operators, and variants thereof [4,10,24,25,27,31,38]. All these strategies guarantee the termination of the *Specialize* procedure. However, since in general the verification problem is undecidable, the assessment of the various generalization strategies, both in terms of precision and verification time, can only be done by an experimental evaluation. That evaluation will be presented in Section 5.

4.1 The *Generalize&Fold* Procedure

The *Generalize&Fold* procedure makes use of a tree of definitions, called *Definition Tree*, whose nodes are labelled by the clauses in $Defs \cup \{\gamma_2\}$. By construction there is a bijection between the set of nodes of the Definition Tree and $Defs \cup \{\gamma_2\}$ and, thus, we will identify each node with its label. The root of the Definition Tree is labelled by clause γ_2 (recall that $\{\gamma_2\}$ is the initial value of *InDefs*) and the children of a clause γ in $Defs \cup \{\gamma_2\}$ are the clauses *NewDefs* derived after applying the procedures $Unfold(\gamma, \Gamma)$ and $Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$. Similarly to [24,25,27,38], our *Generalize&Fold* procedure is based on the combined use of *well-quasi orderings* and clause *generalization* operators. The well-quasi orderings determine when to generalize and guarantee that generalization is eventually applied, while generalization operators determine how to generalize and guarantee that each definition can be generalized a finite number of times only.

Let $\mathcal{C}$ be the set of all constraints and $\mathcal{D}$ be a fixed interpretation for the constraints in $\mathcal{C}$. We assume that: (i) every constraint in $\mathcal{C}$ is a finite conjunction of atomic constraints (conjunction will be denoted by comma), and (ii) $\mathcal{C}$ is closed under projection. The projection of a constraint c onto a tuple of variables X, denoted $project(c, X)$, is a constraint such that $\mathcal{D} \models \forall X\ (project(c, X) \leftrightarrow \exists Y\, c)$, where Y is the tuple of variables occurring in c and not in X. We define a partial order $\sqsubseteq$ on $\mathcal{C}$ as follows: for any two constraints c_1 and c_2 in $\mathcal{C}$, we have that $c_1 \sqsubseteq c_2$ iff $\mathcal{D} \models \forall\ (c_1 \rightarrow c_2)$.

Definition 1 (Well-Quasi Ordering $\preceq$). A *well-quasi ordering* (or *wqo*, for short) on a set S is a reflexive, transitive, binary relation $\preceq$ such that, for every infinite sequence $e_0, e_1, \ldots$ of elements of S, there exist i and j such that $i < j$ and $e_i \preceq e_j$. Given e_1 and e_2 in S, we write $e_1 \approx e_2$ if $e_1 \preceq e_2$ and $e_2 \preceq e_1$. We say that a wqo $\preceq$ is *thin* iff for all $e \in S$, the set $\{e' \in S \mid e \approx e'\}$ is finite.

Definition 2 (Generalization Operator $\ominus$). Let $\preceq$ be a thin wqo on the set $\mathcal{C}$ of constraints. A binary operator $\ominus$ on $\mathcal{C}$ is a *generalization* operator w.r.t. $\preceq$ iff for all constraints c and d in $\mathcal{C}$, we have: (i) $d \sqsubseteq c \ominus d$, and (ii) $c \ominus d \preceq c$. (Note that, in general, $\ominus$ is not commutative.)

The Procedure *Generalize&Fold*

Input: (i) a set *Defs* of definitions, (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure, (iii) a thin wqo $\preceq$, and (iv) a generalization operator $\ominus$ w.r.t. $\preceq$.

Output: (i) A set *NewDefs* of new definitions, and (ii) a set Φ of folded clauses.

NewDefs $:= \emptyset$; $\Phi := \Gamma$;

while in Φ there exists a clause η: $H \leftarrow e, G_1, L, G_2$, where L is either $sat(X, \psi)$ or $\neg sat(X, \psi)$ *do*

GENERALIZE: Let $e_p(X)$ be $project(e, X)$.

1. *if* in *Defs* there exists a clause δ: $newp(X) \leftarrow d(X), sat(X, \psi)$ such that $e_p(X) \sqsubseteq d(X)$ (modulo variable renaming)
   then *NewDefs* := *NewDefs*
2. *elseif* there exists a clause α in *Defs* such that:
   (i) α is of the form $newq(X) \leftarrow b(X), sat(X, \psi)$, and (ii) α is the most recent ancestor of γ in the Definition Tree such that $b(X) \preceq e_p(X)$
   then *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow b(X) \ominus e_p(X), sat(X, \psi)\}$
3. *else* *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow e_p(X), sat(X, \psi)\}$

FOLD: $\Phi := (\Phi - \{\eta\}) \cup \{H \leftarrow e, G_1, M, G_2\}$
where M is $newp(X)$, if L is $sat(X, \psi)$, and M is $\neg newp(X)$, if L is $\neg sat(X, \psi)$

end-while

The following theorem, whose proof is a simple variant of that of Theorem 3 in [17], establishes that the *Specialize* procedure always terminates and preserves the perfect model semantics.

Theorem 2 (Termination and Correctness of the *Specialize* Procedure). *For every input program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, the Specialize procedure terminates. If $P_{\mathcal{S}}$ is the output program of the Specialize procedure, then $P_{\mathcal{S}}$ is stratified* (*and thus, locally stratified*) *and prop $\in M(P_{\mathcal{K}})$ iff prop $\in M(P_{\mathcal{S}})$.*

4.2 Well-Quasi Orderings and Generalization Operators on Linear Constraints

In our verification experiments we will consider the set Lin_k of constraints defined as follows. Every constraint $c \in Lin_k$ is the conjunction of m (≥ 0) *distinct* atomic constraints $a_1, \ldots, a_m$ and, for $i = 1, \ldots, m$, (1) a_i is of the form either $p_i \leq 0$ or $p_i < 0$, and (2) p_i is a polynomial of the form $q_0 + q_1 X_1 + \ldots + q_k X_k$, where $X_1, \ldots, X_k$ are distinct variables and $q_0, q_1, \ldots, q_k$ are integer coefficients.

An equation $r = s$ stands for the conjunction of the two inequations $r \leq s$ and $s \leq r$. In the sequel, when we write $c =_{def} a_1, \ldots, a_m$ we mean that the a_i's are the atomic constraints of c. The constraints in Lin_k are interpreted over the rationals in the usual way.

Well-Quasi Orderings. Now we present three wqo's between constraints in Lin_k, which are based on the integer coefficients of the polynomials. The first wqo is an adaptation to Lin_k of the *homeomorphic embedding* operator [24,25,27,38] and the other two are wqo's based on the maximum and on the sum, respectively, of the absolute values of the coefficients occurring in a constraint.

(W1) The wqo *HomeoCoeff*, denoted by $\precsim_{HC}$, compares sequences of absolute values of integer coefficients occurring in the polynomials. The $\precsim_{HC}$ wqo is based on the notion of homeomorphic embedding and takes into account the commutativity and the associativity of addition and conjunction. Given two polynomials with integer coefficients $p_1 =_{def} q_0 + q_1X_1 + \ldots + q_kX_k$, and $p_2 =_{def} r_0 + r_1X_1 + \ldots + r_kX_k$, we have that $p_1 \precsim_{HC} p_2$ iff there exists a permutation $\langle \ell_0, \ldots, \ell_k \rangle$ of the indexes $\langle 0, \ldots, k \rangle$ such that, for $i = 0, \ldots, k$, $|q_i| \leq |r_{\ell_i}|$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_{HC} a_2$ iff $p_1 \precsim_{HC} p_2$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$ we have that $c_1 \precsim_{HC} c_2$ iff there exist m *distinct* indexes $\ell_1, \ldots, \ell_m$, with $m \leq n$, such that $a_i \precsim_{HC} b_{\ell_i}$, for $i = 1, \ldots, m$.

(W2) The wqo *MaxCoeff*, denoted by $\precsim_{MC}$, compares the maximum absolute value of coefficients occurring in the polynomials. For any atomic constraint a of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, we define $maxcoeff(a)$ to be $max\{|q_0|, |q_1|, \ldots, |q_k|\}$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_{MC} a_2$ iff $maxcoeff(a_1) \leq maxcoeff(a_2)$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_{MC} c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_{MC} b_j$.

(W3) The wqo *SumCoeff*, denoted by $\precsim_{SC}$, compares the sum of the absolute values of the coefficients occurring in the polynomials. For any atomic constraint a of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, , we define $sumcoeff(a)$ to be $\sum_{j=0}^{k} |q_j|$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_{SC} a_2$ iff $sumcoeff(a_1) \leq sumcoeff(a_2)$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_{SC} c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_{SC} b_j$.

The relation $\precsim_{HC}$ is contained both in the relation $\precsim_{MC}$ and in the relation $\precsim_{SC}$. Thus, generalization is applied less often when using $\precsim_{HC}$, instead of $\precsim_{MC}$ or $\precsim_{SC}$. The following table provides some examples of these relations and, in particular, it shows that the relations $\precsim_{MC}$ and $\precsim_{SC}$ are not comparable.

a_1	a_2	$a_1 \precsim_{HC} a_2$	$a_1 \precsim_{MC} a_2$	$a_1 \precsim_{SC} a_2$
$1-2X_1<0$	$3+X_1<0$	yes	yes	yes
$2-2X_1+X_2<0$	$1+3X_1<0$	no	yes	no
$1+3X_1<0$	$2-2X_1+X_2<0$	no	no	yes

Generalization Operators. Now we present some generalization operators on Lin_k which we use in the verification examples of the next section.

(G1) Given any two constraints c and d, the generalization operator *Top*, denoted $\ominus_T$, returns *true*. It can be shown that $\ominus_T$ is indeed a generalization operator with respect to any of the wqo's *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*. The *Top* operator forgets all information about its operands and often determines an over-generalization of the specialized program (see Section 5).

(G2) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and d, the generalization operator *Widen*, denoted $\ominus_W$, returns the constraint $a_{i1}, \ldots, a_{ir}$, such that $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$. Thus, *Widen* keeps all atomic constraints of c that entail d (see [10] for a similar operator used in static program analysis). It can be shown that $\ominus_W$ is indeed a generalization operator with respect to any of the wqo's *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*.

(G3) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and $d =_{def} b_1, \ldots, b_n$, the generalization operator *WidenPlus*, denoted $\ominus_{WP}$, returns the conjunction $a_{i1}, \ldots, a_{ir}, b_{j1}, \ldots, b_{js}$, where: (i) $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$, and (ii) $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim c\}$, where $\precsim$ is a given thin wqo. Thus, *WidenPlus* is similar to *Widen* but, together with the atomic constraints of c that entail d, *WidenPlus* also returns some of the atomic constraints of d. It can be shown that $\ominus_{WP}$ is indeed a generalization operator with respect to both the wqo *MaxCoeff* (in this case $\precsim$ is $\precsim_{MC}$) and the wqo *SumCoeff* (in this case $\precsim$ is $\precsim_{SC}$). However, in general, $\ominus_{WP}$ is *not* a generalization operator with respect to *HomeoCoeff*, because the constraint $c \ominus_{WP} d$ may contain more atomic constraints than c and, thus, it may not be the case that $(c \ominus_{WP} d) \precsim_{HC} c$.

We define our last generalization operator by combining $\ominus_{WP}$ with the *convex hull* operator, which sometimes is used to discover program invariants [10]. The *convex hull* of two constraints c and d in Lin_k, denoted by $ch(c, d)$, is the least (with respect to the $\sqsubseteq$ ordering) constraint h in Lin_k such that $c \sqsubseteq h$ and $d \sqsubseteq h$.

Given a thin wqo $\precsim$ and a generalization operator $\ominus$, we define the generalization operator $\ominus_{CH}$ as follows: for any two constraints c and d, $c \ominus_{CH} d =_{def} c \ominus ch(c, d)$. From the definitions of the operators $\ominus$ and ch one can easily derive that the operator $\ominus_{CH}$ is indeed a generalization operator for c and d, that is, (i) $d \sqsubseteq c \ominus_{CH} d$, and (ii) $c \ominus_{CH} d \precsim c$.

(G4) Given any two constraints c and d, we define the generalization operator *CHWidenPlus*, denoted $\ominus_{CHWP}$, as follows: $c \ominus_{CHWP} d =_{def} c \ominus_{WP} ch(c, d)$.

Note that if we combine the *Top* operator and the convex hull operator, we get again the *Top* operator and, similarly, if we combine the *Widen* operator and the convex hull operator, we get again the *Widen* operator.

The $\sqsubseteq$ ordering on constraints in Lin_k can be extended to an ordering, also denoted $\sqsubseteq$, on the generalization operators, as follows: $\ominus_1 \sqsubseteq \ominus_2$ iff for all constraints c and d, $c \ominus_1 d \sqsubseteq c \ominus_2 d$. For the generalization operators presented above, it can be shown that: (i) $\ominus_{WP} \sqsubseteq \ominus_W \sqsubseteq \ominus_T$, (ii) $\ominus_{CHWP} \sqsubseteq \ominus_W$, and (iii) $\ominus_{WP}$ and $\ominus_{CHWP}$ are not comparable.

The following table shows the application of some generalization operators on constraints of Lin_2 when considering the *MaxCoeff* wqo.

c	$-X_1 \leq 0,\ -2+X_1 \leq 0$	$1-X_1 \leq 0,\ -2+X_1 \leq 0$	$-X_1 \leq 0,\ X_2 \leq 0$
d	$2-X_1 \leq 0,\ 1-X_2 \leq 0$	$-X_1 \leq 0$	$1-X_1 \leq 0,\ -1+X_2 \leq 0$
$c \ominus_W d$	$-X_1 \leq 0$	*true*	$-X_1 \leq 0$
$c \ominus_{WP} d$	$2-X_1 \leq 0,\ 1-X_2 \leq 0$	$-X_1 \leq 0$	$1-X_1 \leq 0,\ -1+X_2 \leq 0$
$c \ominus_{CHWP} d$	$-X_1 \leq 0$	$-X_1 \leq 0$	$-X_1 \leq 0,\ -1+X_2 \leq 0,$ $-X_1+X_2 \leq 0$

5 Experimental Evaluation

In this section we present the results of the experiments we have performed on several examples of verification of infinite state reactive systems. We have implemented the verification algorithm presented in Section 2 using MAP [29], an experimental system for transforming constraint logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `clpq` library to operate on constraints.

We have considered the following *mutual exclusion* protocols and we have verified some of their properties. (i) *Bakery* [14]: we have verified safety (that is, mutual exclusion) and liveness (that is, starvation freedom) in the case of two processes, and safety in the case of three processes; (ii) *MutAst* [23]: we have verified safety in the case of two processes; (iii) *Peterson* [32]: we have verified safety in the case of N (≥ 2) processes by considering a *counting abstraction* [12] of the protocol; and (iv) *Ticket* [14]: we have verified safety and liveness in the case of two processes.

We have also verified safety properties of the following *cache coherence* protocols: (v) *Berkeley RISC*, (vi) *DEC Firefly*, (vii) *IEEE Futurebus+*, (viii) *Illinois University*, (ix) *MESI*, (x) *MOESI*, (xi) *Synapse N+1*, and (xii) *Xerox PARC Dragon*. We have considered the *parameterized* versions of the protocols (v)–(xii) which are designed for an arbitrary number of processors, and we have applied our verification method to the versions derived by using the counting abstraction technique described in [12].

Then we have verified safety properties of the following systems. (xiii) *Barber* [5]: we have considered a parameterized version of this protocol with a single barber process and an arbitrary number of customer processes; (xiv) *Bounded Buffer* and *Unbounded Buffer*: we have considered protocols for two producers and two consumers which communicate via either a bounded or an unbounded buffer, respectively (the encodings of these protocols are taken from [14]); (xv) *CSM*, which is a central server model described in [13]; (xvi) *Insertion Sort* and *Selection Sort*: we have considered the problem of checking array bounds of these two sorting algorithms, parameterized with respect to the size of the array, as presented in [14]; (xvii) *Office Light Control* [39], which is a protocol for controlling how office lights are switched on and off, depending on room occupancy; (xviii) *Reset Petri Nets*, which are Petri Nets augmented with *reset arcs*: we have considered a reachability problem for a net which is a variant of one presented in [26] (unlike [26], we have assumed that in the nets there is no

bound on the number of tokens that can reside in a single place and, therefore, our nets are infinite state systems).

Table 1 shows the results of running the MAP system on the above examples by choosing different combinations of a wqo W and a generalization operator G among those introduced in Section 4. In the sequel we will denote any of these combinations by $W\&G$. The combinations *MaxCoeff&CHWidenPlus*, *MaxCoeff&Top*, and *MaxCoeff&Widen* have been omitted because they give results which are very similar to those obtained by using *SumCoeff*, instead of *MaxCoeff*. We have omitted also the combinations *HomeoCoeff&CHWidenPlus* and *HomeoCoeff&WidenPlus* because, as already mentioned in Section 4, these combinations do not satisfy the conditions given in Definition 2, and thus, they do not guarantee termination of the specialization strategy.

Now we compare the various generalization strategies with respect to *precision* (that is, number of properties proved) and *specialization time* (that is, time taken by the *Specialize* procedure). As expected, we have that precision increases when we consider generalization operators that generalize less (that is, precision is anti-monotonic with respect to the $\sqsubseteq$ relation). Indeed, the use of generalization operators that generalize less, may produce specialized programs that better exploit the knowledge about both the initial state and the property to be proved. In particular, if we use the *SumCoeff* wqo in conjunction with the various generalization operators, then the most precise generalization operator is *WidenPlus* (23 properties proved out of 23), followed by *CHWidenPlus* (22), *Widen* (18), and finally *Top* (17). Similar results are obtained by using the other wqo's introduced in Section 4.

We also have that specialization time increases when we consider generalization operators that generalize less (that is, specialization time is anti-monotonic with respect to the $\sqsubseteq$ relation). This is due to the fact that generalization operators that generalize less may introduce more definitions and, therefore, the specialization phase may take more time. In particular, if we use the *SumCoeff* wqo, then the shortest specialization time is relative to *Top* (the sum of the specialization times of all examples is 1860 ms), followed by *Widen* (4720 ms), *CHWidenPlus* (6280 ms), and finally, *WidenPlus* (12710 ms). Similar results are obtained by using the other wqo's.

Finally, the results of our experiments shown in Table 1 confirm the fact that the use of a wqo for which the *Specialize* procedure performs fewer generalization steps, also determines the introduction of more definitions. Therefore, we have that precision and specialization time increase if we use the *HomeoCoeff* operator, instead of either the *MaxCoeff* operator or the *SumCoeff* operator. In particular, if we compare the two columns of Table 1 for the *Widen* generalization operator we have that: (i) the *Specialize* procedure terminates by using the *HomeoCoeff* wqo if and only if it terminates by using the *SumCoeff* wqo, and (ii) if we consider the examples where the *Specialize* procedure terminates, the sum of the specialization times is 5080 ms for *HomeoCoeff* and 3830 ms for *SumCoeff*. A similar result is obtained by comparing the two columns of Table 1 for the *Top* generalization operator. However, in our examples the increase of precision due to the use of the *HomeoCoeff* wqo, instead of the *SumCoeff* wqo,

was actually confirmed in a weak sense only: for every example the property is verified with *HomeoCoeff&Widen* if and only if it is verified by *SumCoeff& Widen* (and similarly, for *Top*, instead of *Widen*).

In summary, if we consider the tradeoff between precision and verification time, the generalization strategy *SumCoeff&WidenPlus* outperforms the others, closely followed by the generalization strategy *MaxCoeff&WidenPlus*. In particular, the generalization strategies based on either the homeomorphic embedding (that is, *HomeoCoeff*) or the widening and convex hull operators (that is, *Widen* and *CHWidenPlus*) turn out not to be the best strategies in our examples.

In order to compare the implementation of our verification method using MAP with other constraint-based model checking tools for infinite state systems available in the literature, we have performed the verification examples described in Table 1 also on the following systems: (i) ALV [39], which combines BDD-based symbolic manipulation for boolean and enumerated types, with a solver for linear constraints on integers, (ii) DMC [14], which computes (approximated) least and greatest models of CLP(R) programs, and (iii) HyTech [21], a model checker for hybrid systems which handles constraints on reals. All experiments with the MAP, ALV, DMC, and HyTech systems have been performed on an Intel Core 2 Duo E7300 2.66GHz under the operating system Linux. Table 2 reports the results obtained by using various options available in those verification systems.

Table 2 indicates that, in terms of precision, MAP with the *SumCoeff& WidenPlus* option is the best system (23 properties proved out of 23), followed by DMC with the *A* option (19 out of 23), ALV with the *default* option (18 out of 23), and, finally, HyTech with the *Bw* (backward reachability) option (17 out of 23). Among the above mentioned systems, HyTech (*Bw*) has the best average running time (70 ms per proved property), followed by MAP and DMC (both 820 ms), and ALV (8480 ms). This result is explained by the fact that HyTech with the *Bw* option tries to prove a safety property with a very simple strategy, that is, by constructing the reachability set backwards from the property to be proved, while the other systems apply more sophisticated techniques. Note also that the average verification times are affected by the peculiar behaviour on some specific examples. For instance, in the Bounded Buffer and the Unbounded Buffer examples the MAP system has longer verification times with respect to the other systems, because these examples can be easily verified by backward reachability, and this makes the MAP specialization phase unnecessary. On the opposite side, MAP is much more efficient than the other systems in the Peterson N example and the CSM example.

6 Conclusions

In this paper we have proposed some improvements of the method presented in [16] for verifying infinite state reactive systems. First, we have reformulated the verification method as a two-phase method: in Phase (1) a CLP specification of the reactive system is specialized with respect to the initial state and the temporal property to be verified, and in Phase (2) the perfect model of the specialized program is constructed in a bottom-up way. For Phase (1) we

Generalization G:	*WidenPlus*		*CHWidenPlus*	*Widen*		*Top*	
EXAMPLE wqo W:	*MC*	*SC*	*SC*	*HC*	*SC*	*HC*	*SC*
Bakery2 (safety)	30	20	20	20	60	20	40
	30	20	20	20	40	10	30
Bakery2 (liveness)	80	70	80	40	130	60	100
	60	50	60	20	90	40	60
Bakery3 (safety)	180	160	180	150	750	2850	3010
	170	150	170	150	380	670	680
MutAst	70	140	440	110	370	2740	2490
	70	140	420	100	320	310	220
Peterson N	210	230	1370	∞	∞	∞	∞
	210	230	1370	260	250	20	30
Ticket (safety)	20	40	20	20	30	30	30
	10	30	10	10	20	30	20
Ticket (liveness)	110	110	120	90	100	100	110
	60	60	70	50	50	60	60
Berkeley RISC	30	30	200	40	50	20	30
	30	30	170	30	30	10	20
DEC Firefly	30	20	340	140	120	90	80
	30	20	160	60	60	30	20
IEEE Futurebus+	100	2460	47260	47340	47200	15570	15630
	100	270	290	230	230	40	30
Illinois University	40	20	40	80	70	100	90
	30	10	40	50	50	30	30
MESI	30	30	130	40	50	70	70
	30	30	120	30	40	20	20
MOESI	50	60	180	160	160	100	100
	50	50	80	60	60	30	30
Synapse N+1	10	10	10	10	10	20	20
	10	10	10	10	10	10	10
Xerox PARC Dragon	30	40	280	70	70	50	50
	30	40	260	50	50	20	20
Barber	1210	1170	2740	48300	29030	∞	∞
	1170	1130	2620	3090	1620	660	410
Bounded Buffer	3520	3540	6790	340	340	40	20
	2040	2060	6780	140	140	20	20
Unbounded Buffer	3890	3890	410	∞	∞	∞	∞
	360	360	410	110	100	20	10
CSM	6380	6580	4710	∞	∞	∞	∞
	6300	6300	4700	430	440	20	20
Insertion Sort	90	100	160	60	70	100	80
	90	100	150	60	50	30	20
Selection Sort	∞	190	200	∞	∞	∞	∞
	770	180	200	80	70	60	40
Office Light Control	50	50	50	50	50	40	30
	40	40	40	40	40	30	30
Reset Petri Nets	0	0	∞	∞	∞	∞	∞
	0	0	10	0	10	10	10

Table 1. Comparison of various generalization strategies used by the MAP system. For each example we have two lines: the first one shows the *total verification time* (Phases 1 and 2 of our Verification Algorithm) and the second one shows the *program specialization time* (Phase 1 only). *HC*, *MC*, and *SC* denote the wqo *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*, respectively. Times are expressed in milliseconds (ms). The precision is 10 ms. '0' means termination in less than 10 ms. '∞' means no answer within 100 seconds.

EXAMPLE	MAP	ALV				DMC		HyTech	
	SC&WidenPlus	*default*	*A*	*F*	*L*	*noAbs*	*Abs*	*Fw*	*Bw*
Bakery2 (safety)	20	20	30	90	30	10	30	∞	20
Bakery2 (liveness)	70	30	30	90	30	60	70	$\times$	$\times$
Bakery3 (safety)	160	580	570	∞	600	460	3090	∞	360
MutAst	140	$\perp$	$\perp$	910	$\perp$	150	1370	70	130
Peterson N	230	71690	$\perp$	∞	∞	∞	∞	70	∞
Ticket (safety)	40	∞	80	30	∞	∞	60	∞	∞
Ticket (liveness)	110	∞	230	40	∞	∞	220	$\times$	$\times$
Berkeley RISC	30	10	$\perp$	20	60	30	30	∞	20
DEC Firefly	20	10	$\perp$	20	80	50	80	∞	20
IEEE Futurebus+	2460	320	$\perp$	∞	670	4670	9890	∞	380
Illinois University	20	10	$\perp$	∞	140	70	110	∞	20
MESI	30	10	$\perp$	20	60	40	60	∞	20
MOESI	60	10	$\perp$	40	100	50	90	∞	10
Synapse N+1	10	10	$\perp$	10	30	0	0	∞	0
Xerox PARC Dragon	40	20	$\perp$	40	340	70	120	∞	20
Barber	1170	340	$\perp$	90	360	140	230	∞	90
Bounded Buffer	3540	0	10	∞	20	20	30	∞	10
Unbounded Buffer	3890	10	10	40	40	∞	∞	∞	20
CSM	6580	79490	$\perp$	∞	∞	∞	∞	∞	∞
Insertion Sort	100	40	60	∞	70	30	80	∞	10
Selection Sort	190	∞	390	∞	∞	∞	∞	∞	∞
Office Light Control	50	20	20	30	20	10	10	∞	∞
Reset Petri Nets	0	∞	$\perp$	∞	10	0	0	∞	10

Table 2. Comparison of verification times for the MAP, ALV, DMC, and HyTech systems. Times are expressed in milliseconds (ms). The precision is 10 ms. (i) '0' means termination in less than 10 ms. (ii) '$\perp$' means termination with the answer 'unable to verify'. (iii) '∞' means no answer within 100 seconds. (iv) '$\times$' means that the test has not been performed (indeed, HyTech has no built-in for checking liveness). For the MAP system we show the total verification time with the *SumCoeff&WidenPlus* option (see the second column of Table 1). For the ALV system we have four options: *default*, *A* (with approximate backward fixpoint computation), *F* (with approximate forward fixpoint computation), and *L* (with computation of loop closures for accelerating reachability). For the DMC system we have two options: *noAbs* (without abstraction) and *Abs* (with abstraction). For the HyTech system we have two options: *Fw* (forward reachability) and *Bw* (backward reachability).

have considered various specialization strategies which employ different well-quasi orderings and generalization operators to guarantee the termination of the specialization. Some of the well-quasi orderings and generalization operators we use, are adapted from similar notions already known in the area of static analysis of programs [4,10] and program specialization [24,25,27,31,38] (see, in particular, the notions of convex hull, widening, and homeomorphic embedding).

We have applied these specialization strategies to several examples of infinite state systems taken from the literature, and we have compared the results in terms of precision (that is, the number of properties which have been proved) and efficiency (that is, the time taken for the proofs). On the basis of our experimental results we have found that some strategies outperform the others. In particular, in our examples the strategies based on the convex hull, widening, and homeomorphic embedding, do not appear to be the best strategies.

Then, we have applied other tools for the verification of infinite state systems (in particular, ALV [39], DMC [14], and HyTech [21]) to the same set of examples. The experiments show that our specialization-based verification system is quite competitive.

Our approach is closely related to other verification methods for infinite state systems based on the specialization of (constraint) logic programs [26,28,31]. However, unlike the approach proposed in [26,28] we use constraints, which give us very powerful means for dealing with infinite sets of states. The specialization-based verification method presented in [31] consists of one phase only, incorporating top-down query directed specialization and bottom-up answer propagation. That method is restricted to definite constraint logic programs and makes use of a generalization technique which combines widening and convex hull operations for ensuring termination. Unfortunately, in [31] only two examples of verification have been presented (the Bakery protocol and a Petri net) and no verification times are reported and, thus, it is hard to compare that method with our method.

Another approach based on program transformation for verifying parameterized (and, hence, infinite state) systems has been presented in [35]. It is an approach based on unfold/fold transformations which are more general than the ones used by us. However, the strategy for guiding the unfold/fold rules proposed in [35] works in fully automatic mode in a small set of examples only.

Finally, we would like to mention that our verification method can be regarded as complementary to the methods for the verification of infinite state systems based on abstractions [2,4,8,11,14,19,20]. These methods work by constructing approximations of the set of reachable states that satisfy a given property. In contrast, the specialization technique applied during Phase (1) of our method, transforms the program for computing sets of states, but it does not change the set of states satisfying the property of interest. Moreover, during Phase (2) we perform an exact computation of the perfect model of the transformed program.

Further enhancements of infinite state verification could be achieved by combining program specialization and abstraction. In particular, an extension of our method could be done by replacing the bottom-up, exact computation of the

perfect model performed in Phase (2), by an approximated computation in the style of [4,14]. (As already mentioned, this extension would require the computation of both over-approximations and under-approximations of models, because of the presence of negation.) An interesting direction for future research is the study of optimal combinations, both in terms of precision and verification time, of techniques for abstraction and program specialization.

Acknowledgements

We thank the anonymous referees for very valuable comments. We also thank John Gallagher for providing the code of some of the systems considered in Section 5.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. *Proc. LICS'96*, pages 313–321. IEEE Press, 1996.
2. P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction (On efficient verification of parameterized systems). *International Journal of Foundations of Computer Science*, 20(5):779–801, 2009.
3. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
4. G. Banda and J. P. Gallagher. Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. *Proc. LPAR 2010*, LNAI 6355, pages 27–45. Springer, 2010.
5. T. Bultan. BDD vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems. *Proc. TACAS 2000*, LNCS 1785, pages 441–455. Springer, 2000.
6. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.
7. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *Proc. POPL'78*, pages 84–96. ACM Press, 1978.
11. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
12. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
13. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. *Proc. CSL '99*, *LNCS* 1683, pages 50–66. Springer, 1999.
14. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
15. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
16. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proc. VCL'01*, Tech. Rep. DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
17. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying infinite state systems by specializing constraint logic programs. Report 657, IASI-CNR, Roma, Italy, 2007.

18. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. *Proc. CONCUR'97*, LNCS 1243, pages 96–107. Springer, 1997.
19. G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.
20. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR '01*, Lecture Notes in Computer Science 2154, pages 426–440. Springer, 2001.
21. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.
22. K. S. Henriksen, G. Banda, and J. P. Gallagher. Experiments with a convex polyhedral analysis tool for logic programs. In *Proc. WLPE 2007*. CoRR, 2007.
23. D. Lesens and H. Saïdi. Abstraction of parameterized networks. *Electronic Notes of Theoretical Computer Science*, 9:41, 1997.
24. M. Leuschel. Improving homeomorphic embedding for online termination. *Proc. LOPSTR'98*, LNCS 1559, pages 199–218. Springer, 1999.
25. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*, LNCS 2566, pages 379–403. Springer, 2002.
26. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. *Proc. CL 2000*, LNAI 1861, pages 101–115. Springer, 2000.
27. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
28. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. *Proc. LOPSTR'99*, LNCS 1817, pages 63–82. Springer, 2000.
29. The MAP Transformation System. www.iasi.cnr.it/∼proietti/system.html, 2010.
30. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. CL 2000*, LNAI 1861, pages 384–398. Springer, 2000.
31. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. *Proc. LOPSTR 2002*, LNCS 2664, pages 90–108, 2003.
32. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
33. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. *Proc. CAV'96*, LNCS 1102, pages 184–195. Springer, 1996.
34. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *Proc.CAV'97*, LNCS 1254, pages 143–154. Springer, 1997.
35. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. *Proc. TACAS 2000*, LNCS 1785, pages 172–187. Springer, 2000.
36. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
37. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15:49–74, 1999.
38. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. *Proc. ILPS'95*, pages 465–479. MIT Press, 1995.
39. T. Yavuz-Kahveci and T. Bultan. Action Language Verifier: An infinite-state model checker for reactive software specifications. *Formal Methods in System Design*, 35(3):325–367, 2009.