Generalization Strategies for the Verification of Infinite State Systems

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

[1] Dipartimento di Scienze, University 'G. D'Annunzio',
Viale Pindaro 42, I-65127 Pescara, Italy
fioravanti@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[3] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present a comparative evaluation of some generalization strategies which are applied by a method for the automated verification of infinite state reactive systems. The verification method is based on (1) the specialization of the constraint logic program which encodes the system with respect to the initial state and the property to be verified, and (2) a bottom-up evaluation of the specialized program. The generalization strategies are used during the program specialization phase for controlling when and how to perform generalization. Selecting a good generalization strategy is not a trivial task because it must guarantee the termination of the specialization phase itself, and it should be a good balance between precision and performance. Indeed, a coarse generalization strategy may prevent one to prove the properties of interest, while an unnecessarily precise strategy may lead to high verification times. We perform an experimental evaluation of various generalization strategies on several infinite state systems and properties to be verified.

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 (such as n-tuples of integers or rationals) 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, for short), which is a branching time propositional temporal logic by which one can specify, among others, the so-called *safety* and *liveness* properties [9].

Unfortunately, when considering infinite state systems, the verification of CTL formulas is an undecidable problem, in general. In order to cope with this limitation, various *decidable subclasses* of systems and formulas have been identified (see, for instance, [1,14]). Other approaches enhance finite state model

checking by using more general *deductive* techniques (see, for instance, [30,33]) or using *abstractions*, that is, mappings by which one can reduce an infinite state system to a finite state system, preserving the property of interest (see, for instance, [2,35]).

Also logic programming and constraint logic programming 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 [13,27,31]. Moreover, constraints over the integers or the rationals can be used for providing finite representations of infinite sets of states [13,17].

However, for programs which specify infinite state systems, the proof procedures normally used in constraint logic programming (CLP), such as the extension to CLP of SLDNF resolution or tabled resolution [8], 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 using direct program evaluation, many logic programming-based verification methods make use of reasoning techniques such as: (i) static analysis [5,13] or (ii) program transformation [15,23,25,28,32].

In this paper we further develop the verification method presented in [15] 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 [15] as a two-phase method. Phase (1): the CLP specification is specialized w.r.t. the initial state of the system and the temporal property to be verified, and Phase (2): a bottom-up evaluation of the specialized program is performed. By doing so we are able to better evaluate the role of program specialization in the verification technique.

Then, 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. Selecting a good generalization strategy is not a trivial task: it must guarantee the termination of the specialization phase itself, by introducing a finite number of specialization sub-problems, and it should provide a good balance between precision and performance. Indeed, the use of a too coarse generalization strategy may prevent one to prove the properties of interest, while an unnecessarily precise strategy may lead to verification times which are too high. The generalization strategies we consider in this paper have been specifically designed for CLP programs using the constraint domain of linear inequations over rationals.

We have implemented these strategies on the MAP transformation system [26], and we have applied them to several infinite state systems and properties taken from the literature. We have performed a comparative evaluation of generalization strategies in terms of efficiency and power, based on the analysis of the experimental results.

Finally, we have compared our MAP implementation with various constraint-based model checkers for infinite state systems and, in particular, with ALV [7], DMC [13], and HyTech [19].

The paper is organized as follows. In Section 2 we recall how CTL properties of infinite state systems can be encoded by using locally stratified CLP programs. In Section 3 we present our two-phase method for verification. In Section 4 we describe various strategies that can be applied during the specialization phase. These strategies differ for the generalization techniques used for ensuring the termination of the program specialization phase. In Section 5 we report on some experiments we have performed by using a prototype implementation of the MAP transformation system. Finally, we compare the results we have obtained using the MAP system with the results we have obtained using other verification systems.

2 Specifying CTL Properties by CLP Programs

We will model an infinite state system as a *Kripke structure* and we will represent a property to be verified as a formula of the *Computation Tree Logic* (CTL, for short). The fact that a CTL formula φ holds in a state s of a Kripke structure $\mathcal{K}$ will be denoted by $\mathcal{K}, s \models \varphi$. (The reader who is not familiar with these notions may refer to the Appendix or to [9].) A Kripke structure can be encoded as a constraint logic program as indicated in the following four points [15,25,27].
(1) The set S of states is given as a set of n-tuples 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) The set I of initial states is given as a set of clauses of the form:
$initial(X) \leftarrow c(X)$, where $c(X)$ is a constraint.
(3) The transition relation R is given as a set of clauses of the form:
$t(X, Y) \leftarrow c(X, Y)$
where $c(X, Y)$ is a constraint. Y is called a *successor state* of X. We also define 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. We refer to [16] for conditions that guarantee that Ys is a finite list and for an algorithm to construct the clauses defining ts from the clauses defining t.
(4) The elementary properties which are associated with each state X by the labeling function L, are given by a set of clauses of the form:
$elem(X, e) \leftarrow c(X)$
where e is a constant, that is, the name of an elementary property, and $c(X)$ is a constraint.

Given a Kripke structure $\mathcal{K}$ encoded by the clauses defining the predicates $initial$, t, ts, and $elem$, the satisfaction relation $\models$ can be encoded by a predicate sat defined by the following clauses [15,25,27]:

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. $sat\_all([X|Xs], F) \leftarrow sat(X, F), sat\_all(Xs, F)$

Let $P_{\mathcal{K}}$ denote the constraint logic program consisting of clauses 1–10 together with the clauses defining the predicates *initial*, t, ts, and *elem*.

Now we will present a method for proving that a given CTL formula φ holds. We start by defining a new 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:

γ_1 : $prop \leftarrow \neg negprop$

γ_2 : $negprop \leftarrow initial(X), sat(X, not(\varphi))$

The correctness of this encoding is stated by the following Theorem 1 and it is a consequence of the fact that program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is locally stratified and, hence, it has a unique perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ [4]. The proof proceeds by structural induction on the CTL formula φ and is omitted for lack of space (see [16] for details).

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 term of the form $s(X_1, X_2)$ denotes the pair $\langle X_1, X_2 \rangle$ of rationals.

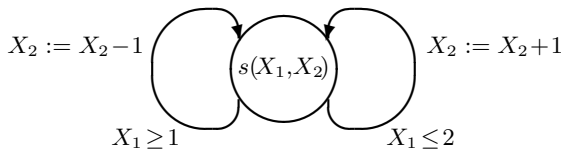

Fig. 1. A reactive system. In any initial state we have that $X_1 \leq 0$ and $X_2 = 0$. No transition changes the value of X_1.

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 the following property: 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 is impossible to reach 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 by the following two clauses:

γ_1: $prop \leftarrow \neg negprop$

γ_2: $negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ sat(s(X_1, X_2), eu(true, negative))$

Thus, by Theorem 1, in order to verify that in every initial state of $\mathcal{K}$ the CTL formula $not(eu(true, negative))$ holds, we need to show that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$. □

One of the most important features of the model checking techniques for finite state systems is the ability to find witnesses and counterexamples [9]. Our encoding of the Kripke structure can easily be extended to handle the generation of witnesses of formulas of the form $eu(\varphi_1, \varphi_2)$ and counterexamples of formulas of the form $af(\varphi)$. For details, the interested reader may refer to Section 7 of [16].

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 indicated in Section 2.

The checking technique based on the bottom-up construction of the perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ is not feasible, in general, because this model may be infinite. Moreover, the proof procedures normally used in constraint logic programming, such as the extension of SLDNF resolution and tabled resolution to CLP, very often diverge when trying to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$. This is due to the limited ability of these proof procedures to cope with infinite failure.

It has been shown in [15] that program specialization often improves the treatment of infinite failure. In this section we present a reformulation of the method proposed in [15] by defining a verification algorithm working in two phases: (1) in the first phase we specialize the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ w.r.t. the query *prop*, that is, w.r.t. initial state of the system and the temporal property to be verified, and (2) in the second phase we construct the perfect model of the specialized program by a bottom-up evaluation. This separation into two phases allows the assessment of the effect of program specialization on the verification process as indicated in Section 5.

It is often the case that the specialization phase improves the termination of the construction of the perfect model. Indeed, in many cases the bottom-up construction of the perfect model of the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ does not terminate because it does not take into account the information about the query to be evaluated and, in particular, about the initial states of the system. The specialization phase modifies the initial program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ by incorporating

into the specialized program P_s the knowledge about the constraints that hold in the initial states. Therefore, the bottom-up construction of the perfect model of P_s may exploit those constraints and terminate more often than the bottom-up construction of the perfect model of the initial program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.

The Verification Algorithm
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.
Output: An Herbrand interpretation M_s such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_s$.
(Phase 1) $Specialize(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}, P_s)$;
(Phase 2) $BottomUp(P_s, M_s)$

The *Specialize* procedure of Phase (1) implements a program specialization strategy which makes use of the following transformation rules only: definition introduction, constrained atomic folding, positive unfolding, constrained atomic folding, removal of clauses with unsatisfiable body, and removal of subsumed clauses. Thus, Phase (1) is simpler than the specialization strategy presented in [15] which uses some extra rules such as negative unfolding, removal of useless clauses, and contextual constraint replacement.

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

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

The *Unfold* procedure takes as input a clause $\gamma \in InDefs$ of the form $H \leftarrow c(X), sat(X, \psi)$, 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 as long as in the body of a clause derived from γ there is an atom of one of the following forms: (i) $t(s_1, s_2)$, (ii) $ts(s, ss)$, (iii) $sat(s, e)$, where e is an elementary property, (iv) $sat(s, not(\psi_1))$, (v) $sat(s, and(\psi_1, \psi_2))$, (vi) $sat(s, ex(\psi_1))$, and (vii) $sat\_all(ss, \psi_1)$, 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 a clause of the form $H \leftarrow c$, where c is a constraint. Due to the structure of the clauses defining the predicates t, ts, sat, and sat_all, the *Unfold* procedure terminates for any ground term denoting a CTL formula ψ occurring in γ.

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, called specialized clauses, which are added to the program P_s. The specialized clauses in Φ are derived as follows. Suppose that η: $H \leftarrow e, L_1, \ldots, L_n$ is a clause in Γ, where for $i = 1, \ldots, n$, L_i is of the form either $sat(X, \psi_i)$ or $\neg sat(X, \psi_i)$. For $i = 1, \ldots, n$, *if* there exists a definition δ_i: $newp(X) \leftarrow d_i(X), sat(X, \psi_i)$ in *Defs* such that e implies $d_i(X)$, *then* γ is folded using δ_i, that is, $sat(X, \psi_i)$ is replaced by $newp(X)$, *else* a new definition ν_i: $newp(X) \leftarrow g(X), sat(X, \psi_i)$, such that e implies $g(X)$, is added to *NewDefs* and γ is folded using ν_i. More details on the *Generalize&Fold* procedure will be given in the next section.

An uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions and, therefore, it may determine the non-termination of the *Specialize* procedure. In order to guarantee termination, we will extend to constraint logic programs some techniques which have been proposed for controlling generalization in *positive supercompilation* [34] and *partial deduction* [22,24].

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, [4]). 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 some standard techniques which are used in the static analysis of programs [10]. Indeed, in order to prove that $prop \in M_s$, we could construct a set $A \subseteq M_s$ such that $prop \in A$. Approximations (also called *abstractions*) are often used for the verification of infinite state systems (see, for instance, [2,5,13,35]). In this paper, however, we will not study these techniques based on approximations and we will focus our attention on the design of the *Specialize* procedure only. In Section 5 we show that the construction of the model M_s terminates in several significant cases.

Example 2. Let us consider the reactive system $\mathcal{K}$ of Example 1. We want to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where *prop* expresses the fact that $not(eu(true, negative))$ holds in every initial state.

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 logic programming system which uses tabled resolution, the query *prop* does not terminate, and (iii) the bottom-up construction of $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ does not terminate (even if we restrict the program to clauses 1, 2, 5, 6, 11–14, which are the clauses actually needed for evaluating the query *prop*).

In our verification algorithm we overcome those difficulties encountered by traditional Prolog systems and XSB 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). By doing so, we derive the following specialized program P_s:

$prop \leftarrow \neg negprop$
$negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ new1(X_1, X_2)$
$new1(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 = 0,\ Y_1 = X_1,\ Y_2 = 1,\ new2(Y_1, Y_2)$
$new2(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 \geq 0,\ Y_1 = X_1,\ Y_2 = X_2 + 1,\ new2(Y_1, Y_2)$

The *BottomUp* procedure computes the perfect model of P_s, which is $M_s = \{prop\}$, in a finite number of steps. Thus, the property $not(eu(true, negative))$ holds in every initial state of $\mathcal{K}$. □

4 Generalization Strategies

The design of a powerful generalization strategy should meet two conflicting requirements: (i) the strategy should enforce the termination of the *Specialize* procedure, and (ii) over-generalization may produce a specialized program P_s with an infinite perfect model which may cause the non-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. All these strategies guarantee the termination of the *Specialize* procedure. However, as we deal with an undecidable verification problem, the power and effectiveness of the various generalization strategies can only be assessed by an experimental evaluation, which will be presented in the next section.

4.1 The *Generalize&Fold* Procedure

The *Generalize&Fold* procedure makes use of a tree, called *Definition Tree*, whose root is labelled by clause γ_2 (recall that $\{\gamma_2\}$ is the initial value of *InDefs*) and the non-root nodes are labelled by the clauses in *Defs*. Since, by construction, the clauses in *Defs* are all distinct, we will identify each clause with a node of that tree. The children of a clause γ in *Defs* are the clauses *NewDefs* derived by applying $Unfold(\gamma, \Gamma)$ followed by $Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$.

Similarly to [22,24,34], our generalization technique is based on the combined use of *well-quasi ordering* relations and clause *generalization* operators. The well-quasi orderings guarantee that generalization is eventually applied, while generalization operators 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 either an atomic constraint or a finite conjunction of constraints (we will denote conjunction by comma), and (ii) $\mathcal{C}$ is closed under projection, where the projection of a constraint c w.r.t. the variable X is a constraint, denoted $project(c, X)$, such that $\mathcal{D} \models \forall(project(c, X) \leftrightarrow \exists X c)$. 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). A *well-quasi ordering* (*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). Let $\preceq$ be a wqo on $\mathcal{C}$. A *generalization* operator on $\mathcal{C}$ w.r.t. the wqo $\preceq$, is a binary operator $\ominus$ such that, 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.)

Similarly to the *widening* operator ∇ used in abstract interpretations [10], every infinite sequence of constraints constructed by using the generalization operator eventually stabilizes, that is, for every infinite sequence $d_0, d_1, \ldots$ of constraints, in the infinite sequence $c_0, c_1, \ldots$ defined as follows:

$c_0 = d_0$ and for any $i \geq 0$, $c_{i+1} = c_i \ominus d_{i+1}$,
there exist $0 \leq m < n$, such that $c_m = c_n$.

Procedure *Generalize&Fold*
Input: (i) a set *Defs* of definitions, and (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure.
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 given in [16], establishes that the *Specialize* procedure, which uses the *Unfold* and the *Generalize&Fold* subprocedures, always terminates and preserves the perfect model semantics.

Theorem 2 (Termination and Correctness of the *Specialize* Procedure). *For every input program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, for every thin wqo $\preceq$, for every generalization operator $\ominus$, the Specialize procedure terminates. If P_s is the output program of the Specialize procedure, then prop $\in M(P_{\mathcal{K}})$ iff prop $\in M(P_s)$.*

4.2 Well-Quasi Orderings and Generalization Operators on Linear Constraints

In this section we describe the wqo's and the generalization operators we have used in our verification experiments.

We will consider the set Lin_k of constraints defined as follows. The atomic constraints of Lin_k are linear inequations constructed by using the predicate symbols $<$ and $\leq$, the k distinct variables $X_1, \ldots, X_k$, and the integer coefficients q_i's. The constraints in Lin_k are interpreted over the rationals in the usual way. Every constraint $c \in Lin_k$ is the conjunction $a_1, \ldots, a_m$ of $m\,(\geq 0)$ *distinct* atomic constraints 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 $q_0 + q_1X_1 + \ldots q_kX_k$, where the q_i's are integer coefficients. An equation $r = s$ is considered as an abbreviation of the conjunction of the two inequations $r \leq s$ and $s \leq r$.

In every infinite state system we will consider, the state is represented as an n-tuple $\langle t_1, \ldots, t_n \rangle$, where k terms, with $k \leq n$, are rationals and the remaining $n-k$ terms are elements of a finite domain. As illustrated in Section 2, the initial states and the elementary properties are specified by constraints in Lin_k and the transition relation is specified by constraints in Lin_{2k}. (The $n-k$ components of a state which are elements of a finite domain, are specified by their values.)

Well-Quasi Orderings. Now we present three wqo's between the polynomials and between the constraints on Lin_k that we will use in the verification examples of the next section.

(1) The wqo *HomeoCoeff*, denoted by $\preceq_{HC}$, compares sequences of absolute values of integer coefficients occurring in polynomials. The $\preceq_{HC}$ ordering is based on the notion of a *homeomorphic embedding* (see, for instance, [22,24,34]) and takes into account commutativity and 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 $p_1 \preceq_{HC} p_2$ iff there exist 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 $a_1 \preceq_{HC} a_2$ iff $p_1 \preceq_{HC} p_2$. Similarly, if we consider $a_1 =_{def} p_1 \leq 0$ 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 $c_1 \preceq_{HC} c_2$ iff there exist m *distinct* indexes $\ell_1, \ldots, \ell_m$, with $m \leq n$, such that $a_i \preceq_{HC} b_{\ell_i}$, for $i = 1, \ldots, m$.

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

(3) The wqo *SumCoeff*, denoted by $\preceq_{SC}$, compares the sum of the absolute values of the coefficients occurring in polynomials. For any atomic constraint a_i of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, , we define $sumcoeff(a_i) = \sum_{j=0}^{k} |q_j|$, and for any two atomic constraints a_1, a_2, we define $a_1 \preceq_{SC} a_2$ iff

sumcoeff$(a_1) \leq$ *sumcoeff*(a_2). Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we define $c_1 \precsim_{SC} c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_{SC} b_j$.

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 w.r.t. the wqo's *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*.

(G2) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and d, the generalization operator *Widen*, denoted $\ominus_W$, returns the conjunction $a_{i1}, \ldots, a_{ir}$, where $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$ (see [10] for a similar operator for static program analysis). It can be shown that $\ominus_W$ is, indeed, a generalization operator w.r.t. 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) $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim c\}$. It can be shown that $\ominus_{WP}$ is, indeed, a generalization operator w.r.t. the wqo's *MaxCoeff*, and *SumCoeff*. However, in general, $\ominus_{WP}$ is *not* a generalization operator w.r.t. *HomeoCoeff*, because the constraint $c \ominus_{WP} d$ may contain more atomic constraints than c and, thus, it may not be the case that $(c \ominus_{WP} d) \precsim_{HC} c$.

Some additional generalization operators can be defined by combining our three operators $\ominus_T$, $\ominus_W$, and $\ominus_{WP}$, with the *convex hull* operator, which is often used in the static analysis of programs [10]. Given two constraints c and d, their *convex hull*, denoted by $ch(c, d)$, is the least constraint h (w.r.t. the $\sqsubseteq$ ordering) such that $c \sqsubseteq h$ and $d \sqsubseteq h$.

Given any two constraints c and d, a wqo $\precsim$, and a generalization operator $\ominus$, we define the generalization operator $\ominus_{CH}$ as follows: $c \ominus_{CH} d =_{def} c \ominus ch(c, d)$. From the definitions of $\ominus$ 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 *CHWiden*, denoted $\ominus_{CHW}$, as follows: $c \ominus_{CHW} d =_{def} c \ominus_W ch(c, d)$.

(G5) 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 generalization operator *Top* and the convex hull operator, we get the *Top* operator again.

5 Experimental Evaluation

In this section we report 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 by using MAP [26], an experimental system for the transformation of constraint

logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `clpq` library to operate on constraints.

Now we give a brief description of the experiments we have conducted (see also Tables 1 and 2).

We have considered the following *mutual exclusion* protocols. (i) *Bakery* [20]: 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* [21]: we have verified safety in the case of two processes; (iii) *Peterson* [29]: we have considered a *counting abstraction* [11] of this protocol and we have verified safety in the case of N processes; and (iv) *Ticket* [3]: we have considered the case of two processes and we have verified safety and liveness.

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*. These protocols are used in shared-memory multiprocessing systems for guaranteeing data consistency of the *local cache* associated with every processor [18]. We have considered *parameterized* versions of the above protocols, that is, protocols designed for an arbitrary number of processors. Similarly to [11], we have applied our verification method to counting abstractions of the protocols.

We have also verified safety properties of the following systems. (xiii) *Barber* [6]: we have considered a parameterized version of this protocol with a single barber process and an arbitrary number of customer processes; (xiv) *Bounded* and *Unbounded Buffer*: we have considered protocols for two producers and two consumers which communicate via a bounded and an unbounded buffer, respectively (the encodings of these protocols are taken from [13]); (xv) *CSM*, which is a central server model described in [12]; (xvi) *Insertion Sort* and *Selection Sort*: we have considered the problem of checking array bounds of these two sorting algorithms, parameterized w.r.t. the size of the array, as presented in [13]; (xvii) *Office Light Control* [7], which is a protocol for controlling how office lights are switched on or 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 [23] (unlike [23], in the net we have considered there is no bound on the number of tokens that can reside in a place and, therefore, our net is an infinite state system).

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 introduced in Section 4. In the sequel we will denote that combination by $W\&G$. The combinations *MaxCoeff*$\&$*CHWiden*, *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*. *HomeoCoeff*$\&$*CHWidenPlus* and *HomeoCoeff*$\&$*WidenPlus* have been omitted because, as already mentioned in Section 4, these combinations do not satisfy the conditions given in Definition 2, and thus, they do not guarantee the termination of the specialization strategy.

If we consider *precision*, that is, the number of properties which have been proved, the best combination is *SumCoeff&WidenPlus* (23 properties proved out of 23) closely followed by *MaxCoeff&WidenPlus* and *SumCoeff&CHWidenPlus* (22 properties proved out of 23). The verification times for *SumCoeff&CHWiden-Plus* are definitely worse than the ones for the other two combinations: indeed, on average, for each proved property, *SumCoeff&CHWidenPlus* takes 2990 milliseconds, while *MaxCoeff&WidenPlus* and *SumCoeff&WidenPlus* take 730 milliseconds and 820 milliseconds, respectively. This is due to the fact that *SumCoeff& CHWidenPlus* introduces more definitions than the other two strategies. (For lack of space, we do not report average times for the other generalization strategies, nor we present statistics on the number of generated definitions.) Table 1 also shows that by using the *Top* and the *Widen* generalization operators the specialization time is quite good, but the precision of verification is low. In other terms, the *Top* and *Widen* operators determine an over-generalization. In contrast, *SumCoeff&WidenPlus* and *MaxCoeff&WidenPlus* ensure a good balance between time and precision.

In conclusion, the specialization strategy which uses the generalization strategy *SumCoeff&WidenPlus* outperforms the others, closely followed by the specialization strategy which uses *MaxCoeff&WidenPlus*. In particular, the generalization strategies based either on the homeomorphic embedding (that is, *Homeo-Coeff*) or the combination of the widening and convex hull operators (that is, *Widen* and *CHWiden*) are not the best choices in our examples.

In order to compare our MAP implementation of the verification method with other constraint-based model checking tools for infinite state systems, we have run the verification examples described above on the following systems: (i) ALV [7], which combines BDD-based symbolic manipulation for boolean and enumerated types, with a solver for linear constraints on integers, (ii) DMC [13], which computes (approximated) least and greatest models of CLP(R) programs, and (iii) HyTech [19], a model checker for hybrid systems.

Table 2 reports the results of our experiments obtained by using various options available in those verification systems. All experiments with MAP, ALV, DMC, and HyTech have been conducted on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system.

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* option (17 out of 23). Among the above mentioned systems and respective options, HyTech (*Bw*) has the best average running time (70 milliseconds per proved property), followed by MAP and DMC (both 820 milliseconds), and ALV (8480 milliseconds). This is explained by the fact that the HyTech with the *Bw* option tries to prove a safety property with a very simple strategy, 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 an uneven behavior on some specific examples. For instance, in the Bounded and Unbounded Buffer examples the

Generalization G:	*CHWiden*		*CHWidenPlus*	*Top*		*Widen*		*WidenPlus*	
EXAMPLE wqo W:	*HC*	*SC*	*SC*	*HC*	*SC*	*HC*	*SC*	*MC*	*SC*
Bakery 2 (safety)	20	70	20	30	40	20	60	30	20
	20	50	20	20	30	20	40	30	20
Bakery 2 (liveness)	60	120	80	80	100	70	130	80	70
	40	80	60	50	60	50	90	60	50
Bakery 3 (safety)	160	800	180	2420	3010	170	750	180	160
	150	430	170	730	680	160	380	170	150
MutAst	230	440	440	2870	2490	220	370	70	140
	200	390	420	330	220	190	320	70	140
Peterson N	∞	∞	1370	∞	∞	∞	∞	210	230
	∞	410	1370	∞	30	∞	250	210	230
Ticket (safety)	30	30	20	20	30	20	30	20	40
	30	20	10	20	20	20	20	10	30
Ticket (liveness)	90	120	120	100	110	100	100	110	110
	50	70	70	60	60	60	50	60	60
Berkeley RISC	60	60	200	70	30	50	50	30	30
	60	40	170	50	20	50	30	30	30
DEC Firefly	190	120	340	100	80	180	120	30	20
	100	60	160	40	20	90	60	30	20
IEEE Futurebus+	∞	47260	47260	∞	15630	∞	4720	100	2460
	∞	290	290	∞	30	∞	230	100	270
Illinois University	50	80	40	140	90	50	70	40	20
	50	60	40	60	30	50	50	30	10
MESI	100	50	130	100	70	100	50	30	30
	80	40	120	50	20	80	40	30	30
MOESI	980	160	180	930	100	940	160	50	60
	950	60	80	860	30	910	60	50	50
Synapse N+1	30	10	10	20	20	20	10	10	10
	20	10	10	20	10	10	10	10	10
Xerox PARC Dragon	1230	80	280	1140	50	1210	70	30	40
	1180	60	260	1110	20	1160	50	30	40
Barber	41380	30150	2740	∞	∞	40750	29030	1210	1170
	3260	3100	2620	900	410	2630	1620	1170	1130
Bounded Buffer	73990	370	6790	71870	20	75330	340	3520	3540
	73190	170	6780	71850	20	74550	140	2040	2060
Unbounded Buffer	∞	∞	410	∞	∞	∞	∞	3890	3890
	310	130	410	140	10	280	100	360	360
CSM	∞	∞	4710	∞	∞	∞	∞	6380	6580
	∞	620	4700	30	20	∞	440	6300	6300
Insertion Sort	80	80	160	110	80	70	70	90	100
	80	60	150	30	20	70	50	90	100
Selection Sort	∞	∞	200	∞	∞	∞	∞	∞	190
	380	80	200	40	40	340	70	770	180
Office Light Control	40	50	50	40	30	50	50	50	50
	30	40	40	30	30	40	40	40	40
Reset Petri Nets	∞	∞	∞	∞	∞	∞	∞	0	0
	10	10	10	0	10	0	10	0	0

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

MAP system has higher verification times with respect to the other systems, because these examples can be easily verified by backward reachability, thereby making the MAP specialization phase redundant. On the opposite side, MAP is much more efficient than the other systems in the Peterson and CSM examples, where the specialization phase definitely pays off.

6 Conclusions

In this paper we have proposed some improvements of the method presented in [15] for verifying infinite state reactive systems. First, we have reformulated the verification method as a two-phase method: (1) in the first phase a CLP specification of the reactive system is specialized w.r.t. the initial state and the temporal property to be verified and (2) in the second phase the perfect model of the specialized program is constructed in a bottom-up way. For Phase (1) we have considered various specialization strategies which employ different well-quasi orderings and generalization operators to guarantee always terminating transformations. These orderings and generalization operators are either new or adapted from similar notions, such as, convex hull, widening, and homeomorphic embedding, defined in the context of static analysis of programs [10] and program specialization [22,24,34].

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 efficiency and precision (that is, the number of proved properties). On the basis of our experimental results we have found some strategies that outperform the others in terms of a good balance of efficiency and precision. In particular, the strategies based on the convex hull, widening, and homeomorphic embedding, do not appear to be the best strategies in our examples.

Then, we have applied other model checking tools for infinite state systems (in particular, ALV [7], DMC [13], and HyTech [19]) to the same set of examples. The experimental results show that the specialization-based verification system (with the best performing strategies) is competitive with respect to the other tools.

Our approach is closely related to other verification methods for infinite state systems based on the specialization of (constraint) logic programs [23,25,28]. Unlike the approach proposed in [23,25] we use constraints, which give us very powerful means of dealing with infinite sets of states. The specialization-based verification method presented in [28] consists of one phase only incorporating both top-down query directed specialization and bottom-up answer propagation. This method is restricted to definite constraint logic programs and makes use of a generalization technique which combines widening and convex hull computations for ensuring termination. In [28] only two examples have been presented (the Bakery protocol and a Petri net) and no verification times are reported. Thus, it is hard to make a comparison in terms of effectiveness with respect to the method we have presented here. However, as already mentioned, the generalization techniques based on the widening and the convex hull operators do not turn out to be the best options in our examples.

EXAMPLE	MAP	ALV				DMC		HyTech	
	SC&WidenPlus	*default*	*A*	*F*	*L*	*noAbs*	*Abs*	*Fw*	*Bw*
Bakery 2 (safety)	20	20	30	90	30	10	30	∞	20
Bakery 2 (liveness)	70	30	30	90	30	60	70	$\times$	$\times$
Bakery 3 (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 (livenes)	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
Unbonded 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 the MAP system with the verification systems ALV, DMC, and HyTech. Times are expressed in milliseconds. The precision is 10 milliseconds. (i) '0' means termination in less than 10 milliseconds. (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 last column of Table 1). For the ALV system we show the time for 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 show the time for two options: *noAbs* (without abstraction) and *Abs* (with abstraction). For the HyTech system we show the time for two options: *Fw* (forward reachability) and *Bw* (backward reachability).

Another approach based on program transformation for verifying parameterized (and, hence, infinite state) systems has been presented in [32]. This approach is based on unfold/fold transformations which are more general than the ones used in the specialization strategy considered in this paper. However, the strategy for guiding the unfold/fold rules proposed in [32] works in fully automatic mode in a small set of examples only.

Finally, we would like to mention that our verification method can be viewed as complementary with respect to the methods based on static analysis, such as [5,13]. These methods work by constructing approximations of program models (which can be least or greatest models, depending on the specific technique) in a bottom-up way. However, these methods may fail to prove a property simply because they compute a subset (or a superset) of the program model.

One further enhancement of our method could be achieved by replacing the bottom-up, precise computation of the perfect model performed in our Phase (2) by an approximated computation, in the style of [5,13]. Finding the optimal combination, in terms of both efficiency and precision, of program specialization and static analysis techniques for the verification of infinite state systems is an interesting direction for future research.

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. G. R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.
4. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
5. G. Banda and J. P. Gallagher. Constraint-based abstraction of a model checker for infinite state systems. *Proc. WLP 2009*. Potsdam, Germany, 2009.
6. 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.
7. T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. *Proc. ASE 2001*, pages 382–386. IEEE Press, 2001.
8. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
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. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
12. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. *Proc. CSL '99*, *LNCS* 1683, pages 50–66. Springer, 1999.
13. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

14. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
15. 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.
16. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying infinite state systems by specializing constraint logic programs. Report 657, IASI-CNR, Roma, Italy, 2007.
17. 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.
18. J. Handy. *The Cache Memory Book*. Morgan Kaufman, 1998. Second Edition.
19. 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.
20. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
21. D. Lesens and H. Saïdi. Abstraction of parameterized networks. *Electronic Notes of Theoretical Computer Science*, 9:41, 1997.
22. M. Leuschel. Improving homeomorphic embedding for online termination. *Proc. LOPSTR'98*, LNCS 1559, pages 199–218. Springer, 1999.
23. 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.
24. 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.
25. 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.
26. The MAP Transformation System. www.iasi.cnr.it/∼proietti/system.html, 2010.
27. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. CL 2000*, LNAI 1861, pages 384–398. Springer, 2000.
28. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of clp programs. *Proc. LOPSTR 2002*, LNCS 2664, pages 90–108, 2003.
29. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
30. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. *Proc. CAV'96*, LNCS 1102, pages 184–195. Springer, 1996.
31. 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.
32. 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.
33. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15:49–74, 1999.
34. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. *Proc. ILPS'95*, pages 465–479. MIT Press, 1995.
35. L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (A survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.

Appendix

A *Kripke structure* [9] is a 4-tuple $\langle S, I, R, L\rangle$, where: (1) S is a set of *states*, (2) $I \subseteq S$ is a set of *initial states,* (3) $R \subseteq S \times S$ is a *transition relation* from states to states, and (4) L: $S \rightarrow \mathcal{P}(Elem)$ is a *labeling function* that assigns to each state $s \in S$ a subset $L(s)$ of *Elem*, that is, a set of elementary properties that hold in s. In particular, there exist the elementary properties *true*, *false*, and *initial* and we have that, for all $s \in S$, (i) $true \in L(s)$, (ii) $false \notin L(s)$, and (iii) $initial \in L(s)$ iff $s \in I$. We assume that R is a *total* relation, that is, for every state $s \in S$ there exists at least one state $s' \in S$, called a *successor state* of s, such that $s\, R\, s'$.

A *computation path* in a Kripke structure $\mathcal{K}$ is an *infinite* sequence of states $s_0\, s_1 \ldots$ such that $s_i\, R\, s_{i+1}$ for every $i \geq 0$.

The *Computation Tree Logic* (CTL, for short) [9] is a propositional temporal logic interpreted over a given Kripke structure. A CTL formula φ has the following syntax:

$\varphi ::= e \mid not(\varphi) \mid and(\varphi_1, \varphi_2) \mid ex(\varphi) \mid eu(\varphi_1, \varphi_2) \mid af(\varphi)$

where e belongs to the set *Elem* of the elementary properties. Note that the set $\{ex, eu, af\}$ of operators is sufficient to express all CTL properties, because the other CTL operators introduced in [9] can be defined in terms of ex, eu, and af. In particular, the operator ef is defined as $eu(true, \varphi)$.

The operators ex, eu, and af have the following semantics. The property $ex(\varphi)$ holds in a state s if there exists a successor s' of s such that φ holds in s'. The property $eu(\varphi_1, \varphi_2)$ holds in a state s if there exists a computation path π starting from s such that φ_1 holds in all states of a finite prefix of π and φ_2 holds on the rest of the path. The property $af(\varphi)$ holds in a state s if on every computation path π starting from s there exists a state s' where φ holds.

Formally, the semantics of CTL formulas is given by defining the satisfaction relation $\mathcal{K}, s \models \varphi$, which tells us when a formula φ holds in a state s of the Kripke structure $\mathcal{K}$.

Definition 3 (Satisfaction Relation for a Kripke Structure). Given a Kripke structure $\mathcal{K} = \langle S, I, R, L\rangle$, a state $s \in S$, and a formula φ, we inductively define the relation $\mathcal{K}, s \models \varphi$ as follows:

$\mathcal{K}, s \models e$ iff e is an elementary property belonging to $L(s)$

$\mathcal{K}, s \models not(\varphi)$ iff it is not the case that $\mathcal{K}, s \models \varphi$

$\mathcal{K}, s \models and(\varphi_1, \varphi_2)$ iff $\mathcal{K}, s \models \varphi_1$ and $\mathcal{K}, s \models \varphi_2$

$\mathcal{K}, s \models ex(\varphi)$ iff there exists a computation path $s_0\, s_1 \ldots$ in $\mathcal{K}$ such that $s = s_0$ and $\mathcal{K}, s_1 \models \varphi$

$\mathcal{K}, s \models eu(\varphi_1, \varphi_2)$ iff there exists a computation path $s_0\, s_1 \ldots$ in $\mathcal{K}$ such that (i) $s = s_0$ and (ii) for some $n \geq 0$ we have that: $\mathcal{K}, s_n \models \varphi_2$ and $\mathcal{K}, s_j \models \varphi_1$ for all $j \in \{0, \ldots, n-1\}$

$\mathcal{K}, s \models af(\varphi)$ iff for all computation paths $s_0\, s_1 \ldots$ in $\mathcal{K}$, if $s = s_0$ then there exists $n \geq 0$ such that $\mathcal{K}, s_n \models \varphi$. □