Controlling Polyvariance for Specialization-Based Verification

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[3], and Valerio Senni[2,4]

[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
[4] LORIA-INRIA, Villers-les-Nancy, France
valerio.senni@loria.fr

Abstract. We present some extensions of a method for verifying safety properties of infinite state reactive systems. Safety properties are specified by constraint logic programs encoding (backward or forward) reachability algorithms. These programs are transformed, before their use for checking safety, by specializing them with respect to the initial states (in the case of backward reachability) or with respect to the unsafe states (in the case of forward reachability). In particular, we present a specialization strategy which is more general than previous proposals and we show, through some experiments performed on several infinite state reactive systems, that by using the specialized reachability programs obtained by our new strategy, we considerably increase the number of successful verifications. Then we show that the specialization time, the size of the specialized program, and the number of successful verifications may vary, depending on the *polyvariance* introduced by the specialization, that is, the set of specialized predicates which have been introduced. Finally, we propose a general framework for controlling polyvariance and we use our set of examples of infinite state reactive systems to compare in an experimental way various control strategies one may apply in practice.

1 Introduction

Program specialization is a program transformation technique that, given a program and a specific context of use, derives a specialized program that is more effective in the given context [19]. Program specialization techniques have been proposed for several programming languages and, in particular, for (constraint) logic languages (see, for instance [7,11,16,17,21,22,24,27]).

Program specialization may generate *polyvariant procedures*, that is, it may derive, starting from a single procedure, multiple specialized versions of that procedure. In the case of (constraint) logic programming, program specialization may introduce several new predicates corresponding to specialized versions of a predicate occurring in the original program. The application of specialized procedures to specific input values often results in a very efficient computation. However, if the number of new predicate definitions and, hence, the size of the specialized program, is overly large, we may have difficulties during program compilation and execution.

In order to find an optimal balance between the degree of specialization and the size of the specialized program, several papers have addressed the issue of *controlling* polyvariance (see [22,26], in the case of logic programming). This issue is related to the one of controlling *generalization* during program specialization, because a way of reducing unnecessary polyvariance is to replace several specialized procedures by a single, more general one.

In this paper we address the issue of controlling polyvariance in the context of specialization-based techniques for the automatic verification of properties of reactive systems [12,13,23].

One of the present challenges in the verification field is the extension of model checking techniques [5] to systems with an infinite number of states. For these systems exhaustive state exploration is impossible and, even for restricted classes, simple properties such as *safety* (or *reachability*) properties are undecidable (see [9] for a survey of relevant results).

In order to overcome this limitation, several authors have advocated the use of *constraints* to represent infinite sets of states and constraint logic programs to encode temporal properties (see, for instance, [8,15]). By using constraint-based methods, a temporal property can be verified by computing the least or the greatest models of programs, represented as finite sets of constraints. Since, in general, the computation of these models may not terminate, various techniques have been proposed based on *abstract interpretation* [2,3,6,8] and *program specialization* [12,13,23].

The techniques based on abstract interpretation compute a conservative approximation of the program model, which is sometimes sufficient to prove that the property of interest actually holds. However, in the case where the property does not hold in the approximated model, one cannot conclude that the property does not hold.

The techniques based on program specialization transform the program that encodes the property of interest by taking into account the property to be proved and the initial states of the system, so that the construction of the model of the transformed program may terminate more often than the one of the original program, that is, the so-called *verification precision* is improved.

In this paper we show that the control of polyvariance plays a very relevant role in verification techniques based on program specialization. Indeed, the specialization time, the size of the specialized program, and the precision of verification may vary depending on the set of specialized predicates introduced by different specialization strategies. We also propose a general framework for controlling polyvariance during specialization and, through several examples of infinite state reactive systems taken from the verification literature, we compare in an experimental way various control strategies that may be applied in practice.

Our paper is structured as follows. In Section 2 we present a method based on constraint logic programming for specifying and verifying safety properties of infinite state reactive systems. In Sections 3 and 4 we present a general framework for specializing constraint logic programs that encode safety properties of infinite state reactive systems and, in particular, for controlling polyvariance during specialization. In Section 5 we present some experimental results. Finally, in Section 6 we compare our method with related approaches in the field of program specialization and verification.

2 Specialization-Based Reachability Analysis of Infinite State Reactive Systems

An infinite state reactive system is specified as follows. A *state* is an n-tuple $\langle a_1, \ldots, a_n \rangle$ where each a_i is either an element of a finite domain $\mathbb{D}$ or an element of the set $\mathbb{R}$ of the real numbers. By X we denote a variable ranging over states, that is, an n-tuple of variables $\langle X_1, \ldots, X_n \rangle$ where each X_i ranges over either $\mathbb{D}$ or $\mathbb{R}$. Every constraint c is a (possibly empty) conjunction $fd(c)$ of equations on a finite domain $\mathbb{D}$ and a (possibly empty) conjunction $re(c)$ of linear inequations on $\mathbb{R}$. An equation on $\mathbb{R}$ is considered as a conjunction of two inequations. Given a constraint c, every equation in $fd(c)$ and every linear inequation in $re(c)$ is said to be an *atomic constraint*.

The set I of the *initial states* is represented by a disjunction $init_1(X) \vee \ldots \vee init_k(X)$ of constraints on X. The *transition relation* is a disjunction $t_1(X, X') \vee \ldots \vee t_m(X, X')$ of constraints on X and X', where X' is the n-tuple $\langle X'_1, \ldots, X'_n \rangle$ of primed variables.

A constraint c is also denoted by $c(X)$, when we want indicate that the variable X occurs in it. Similarly, for constraints denoted by $c(X')$ or $c(X, X')$. Given a constraint c and a tuple V of variables, we define the *projection* $c|_V$ to be the constraint d such that: (i) the variables of d are among the variables in V, and (ii) $\mathbb{D} \cup \mathbb{R} \models d \leftrightarrow \exists Z\ c$, where Z is the tuple of the variables occurring in c and not in V. We assume that the set of constraints is closed under projection.

Given a clause C of the form $H \leftarrow c \wedge G$, by $con(C)$ we denote the constraint c. A clause of the form $H \leftarrow c$, where c is a constraint, is said to be a *constrained fact*. We say that a constrained fact $H \leftarrow c$ *subsumes* a clause $H \leftarrow d \wedge G$, where d is a constraint and G is a goal, iff d *entails* c, written $d \sqsubseteq c$, that is, $\mathbb{D} \cup \mathbb{R} \models \forall(d \rightarrow c)$.

In this paper we will focus on the verification of *safety* properties. A safety property holds iff *an unsafe state cannot be reached from an initial state of the system.* The set U of the unsafe states is represented by a disjunction $u_1(X) \vee \ldots \vee u_n(X)$ of constraints.

One can verify a safety property by one of the following two strategies:
(i) the *Backward Strategy*: one applies a *backward reachability* algorithm, thereby computing the set BR of states from which it is possible to reach an *unsafe* state, and then one checks whether or not BR has an empty intersection with the set I of the initial states;

(ii) the *Forward Strategy*: one applies a *forward reachability* algorithm, thereby computing the set FR of states reachable from an initial state, and then one checks whether or not FR has an empty intersection with the set U of the unsafe states.

Variants of these two strategies have been proposed and implemented in various automatic verification tools [1,4,14,20,28].

The Backward and Forward Strategies can easily be encoded into constraint logic programming. In particular, we can encode the backward reachability algorithm by means of the following constraint logic program Bw:

$$
\begin{aligned}
I_1&: \mathit{unsafe} \leftarrow \mathit{init}_1(X) \wedge \mathit{bwReach}(X)\\
&\ \cdots\\
I_k&: \mathit{unsafe} \leftarrow \mathit{init}_k(X) \wedge \mathit{bwReach}(X)\\
T_1&: \mathit{bwReach}(X) \leftarrow t_1(X, X') \wedge \mathit{bwReach}(X')\\
&\ \cdots\\
T_m&: \mathit{bwReach}(X) \leftarrow t_m(X, X') \wedge \mathit{bwReach}(X')\\
U_1&: \mathit{bwReach}(X) \leftarrow u_1(X)\\
&\ \cdots\\
U_n&: \mathit{bwReach}(X) \leftarrow u_n(X)
\end{aligned}
$$

We have that: (i) $\mathit{bwReach}(X)$ holds iff an unsafe state can be reached from the state X in zero or more applications of the transition relation, and (ii) *unsafe* holds iff there exists an initial state of the system from which an unsafe state can be reached.

The semantics of program Bw is given by the *least model*, denoted $M(Bw)$, that is, the set of ground atoms derived by using: (i) the theory of equations over the finite domain $\mathbb{D}$ and the theory of linear inequations over the reals $\mathbb{R}$ for the evaluation of the constraints, and (ii) the usual least model construction (see [18] for more details).

The system is *safe* if and only if $\mathit{unsafe} \notin M(Bw)$.

Example 1. Let us consider an infinite state reactive system where each state is a pair of real numbers and the following holds:
(i) the set of initial states is the set of pairs $\langle X_1, X_2\rangle$ such that the constraint $X_1 \geq 1 \wedge X_2 = 0$ holds;
(ii) the transition relation is the set of pairs of states $\langle\langle X_1, X_2\rangle, \langle X_1', X_2'\rangle\rangle$ such that the constraint $X_1' = X_1 + X_2 \wedge X_2' = X_2 + 1$ holds; and
(iii) the set of unsafe states is the set of pairs $\langle X_1, X_2\rangle$ such that the constraint $X_2 > X_1$ holds.
For the above system the predicate *unsafe* is defined by the following CLP program $Bw1$:

1. $\mathit{unsafe} \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge \mathit{bwReach}(X_1, X_2)$
2. $\mathit{bwReach}(X_1, X_2) \leftarrow X_1' = X_1 + X_2 \wedge X_2' = X_2 + 1 \wedge \mathit{bwReach}(X_1', X_2')$
3. $\mathit{bwReach}(X_1, X_2) \leftarrow X_2 > X_1$ □

The Backward Strategy can be implemented by the bottom-up construction of the least fixpoint of the *immediate consequence operator* S_{Bw}, that is, by computing $S_{Bw}\!\uparrow\!\omega$ [18]. The operator S_{Bw} is analogous to the usual immediate consequence operator associated with logic programs, but constructs a set of constrained facts, instead of a set of ground atoms. We have that $M(Bw)$ is the set of ground atoms of the form $A\vartheta$ such that there exists a constrained fact $A \leftarrow c$ in $S_{Bw}\!\uparrow\!\omega$ and the constraint $c\vartheta$ is satisfiable. BR is the set of all states s such that there exists a constrained fact of the form $\mathit{bwReach}(X) \leftarrow c(X)$ in $S_{Bw}\!\uparrow\!\omega$ and $c(s)$ holds. Thus, by using clauses $I_1, \ldots, I_k$, we have that the atom *unsafe* holds iff $BR \cap I \neq \emptyset$.

One weakness of the Backward Strategy is that, when computing BR, it does not take into account the constraints holding on the initial states. This may lead to a failure of the verification process, even if the system is safe, because the computation of $S_{Bw}\uparrow\omega$ may not terminate. A similar weakness is also present in the Forward Strategy as it does not take into account the properties holding on the unsafe states when computing FR.

In this paper we present a method, based upon the program specialization technique introduced in [13], for overcoming these weaknesses. For reasons of space we will present the details of our method for the Backward Strategy only. The application of our method in the case of the Forward Strategy is similar, and we will briefly describe it when presenting our experimental results in Section 5.

The objective of program specialization is to transform the constraint logic program Bw into a new program $SpBw$ such that: (i) $\mathit{unsafe} \in M(Bw)$ iff $\mathit{unsafe} \in M(SpBw)$, and (ii) the computation of $S_{SpBw} \uparrow \omega$ terminates more often than $S_{Bw} \uparrow \omega$ because it exploits the constraints holding on the initial states.

Let us show how our method based program specialization works on the infinite state reactive system of Example 1.

Example 2. Let us consider the program $Bw1$ of Example 1. The computation of $S_{Bw1}\!\uparrow\!\omega$ does not terminate, because it does not take into account the information about the set of initial states, represented by the constraint $X_1 \geq 1 \wedge X_2 = 0$. (One can also check that the top-down evaluation of the query *unsafe* does not terminate either.)

This difficulty can be overcome by specializing the program $Bw1$ with respect to the constraint $X_1 \geq 1 \wedge X_2 = 0$. Similarly to [13], we apply a specialization technique based on the *unfolding* and *folding* transformation rules for constraint logic programs (see, for instance, [10]). We introduce a new predicate $new1$ defined as follows:

4. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge \mathit{bwReach}(X_1, X_2)$

We fold clause 1 using clause 4, that is, we replace the atom $\mathit{bwReach}(X_1, X_2)$ by $new1(X_1, X_2)$ in the body of clause 1, and we get:

5. $\mathit{unsafe} \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge new1(X_1, X_2)$

Now we continue the transformation from the definition of the newly introduced predicate $new1$. We unfold clause 4, that is, we replace the occurrence of $\mathit{bwReach}(X_1, X_2)$ by the bodies of the clauses 2 and 3 defining $\mathit{bwReach}(X_1, X_2)$ in $Bw1$, and we derive:

6. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge X_1' = X_1 \wedge X_2' = 1 \wedge \mathit{bwReach}(X_1', X_2')$

In order to fold clause 6 we may use the following definition, whose body consists (modulo variable renaming) of the atom $bwReach(X'_1, X'_2)$ and the constraint $X_1 \geq 1 \wedge X_2 = 0 \wedge X'_1 = X_1 \wedge X'_2 = 1$ projected w.r.t. the variables $\langle X'_1, X'_2 \rangle$:

7. $newp(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 1 \wedge bwReach(X_1, X_2)$

However, if we repeat the process of unfolding and, in order to fold, we introduce new predicate definitions whose bodies consist of the atom $bwReach(X'_1, X'_2)$ and projected constraints w.r.t. $\langle X'_1, X'_2 \rangle$, then we will introduce, in fact, an infinite sequence of new predicate definitions of the form:

$$newq(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = k \wedge bwReach(X_1, X_2)$$

where k gets the values $1, 2, \ldots$ In order to terminate the specialization process we apply a *generalization strategy* and we introduce the following predicate definition which is a generalization of both clauses 4 and 7:

8. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 \geq 0 \wedge bwReach(X_1, X_2)$

We fold clause 6 using clause 8 and we get:

9. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge X'_1 = X_1 \wedge X'_2 = 1 \wedge new2(X'_1, X'_2)$

Now we continue the transformation from the definition of the newly introduced predicate $new2$. By unfolding clause 8 and then folding using again clause 8 we derive:

10. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 \geq 0 \wedge X'_1 = X_1 + X_2 \wedge X'_2 = X_2 + 1 \wedge new2(X'_1, X'_2)$
11. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 > X_1$

The final specialized program, called $SpBw1$, is made out of clauses 5, 9, 10, and 11. Now the computation of $S_{SpBw1} \uparrow \omega$ terminates due to the presence of the constraint $X_1 \geq 1$ which holds on the initial states and occurs in all clauses of $SpBw1$. □

The form of the specialized program strongly depends on the strategy used for introduction of new predicates corresponding to the specialized versions of the predicate *bwReach*. For instance, in Example 1 we have introduced the two new predicates $new1$ and $new2$, and then we have obtained the specialized program by deriving mutually recursive clauses defining those predicates. Note, however, that the definition of $new2$ is *more general than* the definition of $new1$, because the constraint occurring in the body of the clause defining $new1$ implies the constraint occurring in the body of the clause defining $new1$. Thus, by applying an alternative strategy we could introduce $new2$ only and derive a program $SpBw2$ where clauses 5 and 9 are replaced by the following clause:

12. $unsafe \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge new2(X_1, X_2)$

Program $SpBw2$ is smaller than $SpBw1$ and the computation of $S_{SpBw2} \uparrow \omega$ terminates in fewer steps than the one of $S_{SpBw1} \uparrow \omega$.

In general, when applying our specialization-based verification method there is an issue of *controlling polyvariance*, that is, of introducing a set of new predicate definitions that perform well with respect to the following objectives:
(i) ensuring the termination and the efficiency of the specialization strategy,
(ii) minimizing the size of the specialized program, and

(iii) ensuring the termination and the efficiency of the fixpoint computation of the least models.

The objective of ensuring the termination of the fixpoint computation (and, thus, the *precision* of the verification) can be in contrast with the other objectives, because it may need the introduction of less general predicates, while the achievement of other objectives is favoured by the introduction of more general predicates. In the next section we will present a framework for controlling polyvariance and achieving a good balance between the requirements we have listed above.

3 A Generic Algorithm for Controlling Polyvariance During Specialization

The core of our technique for controlling polyvariance is an algorithm for specializing the CLP program *Bw* with respect to the constraints characterizing the set of initial states. Our algorithm is *generic*, in the sense that it depends on three unspecified procedures: (1) *Partition*, (2) *Generalize*, and (3) *Fold*. Various definitions of the *Partition*, *Generalize*, and *Fold* procedures will be given in the next section, thereby providing concrete specialization algorithms. These definitions encode techniques already proposed in the specialization and verification fields (see, for instance, [6,13,22,27]) and also new techniques proposed in this paper.

Our generic specialization algorithm (see Figure 1) constructs a tree, called *DefsTree*, where: (i) each node is labelled by a clause of the form $newp(X) \leftarrow d(X) \wedge bwReach(X)$, called a *definition*, defining a new predicate introduced during specialization, and (ii) each arc from node D_i to node D_j is labelled by a subset of the clauses obtained by unfolding the definition of node D_i. When no confusion arises, we will identify a node with its labelling definition. An arc from definition D_i to definition D_j labelled by the set *Cs* of clauses is denoted by $D_i \xrightarrow{Cs} D_j$.

The definition at the root of *DefsTree* is denoted by the special symbol T. Initially, *DefsTree* is $\{\mathsf{T} \xrightarrow{\{I_1\}} D_1, \ldots, \mathsf{T} \xrightarrow{\{I_k\}} D_k\}$, where (i) $I_1, \ldots, I_k$ are the clauses defining the predicate *unsafe* in program *Bw* (see Section 2), and (ii) for $j = 1, \ldots, k$, D_j is the clause $new_j(X) \leftarrow init_j(X) \wedge bwReach(X)$, such that new_j is a new predicate symbol and the body of D_j is equal to the body of I_j.

A definition D in *DefsTree* is said to be *recurrent* iff D labels both a leaf node and a non-leaf node of *DefsTree*.

We construct the children of a non-recurrent definition D in the definition tree *DefsTree* constructed so far, as follows. We unfold D with respect to the atom $bwReach(X)$ occurring in its body, that is, we replace $bwReach(X)$ by the bodies of the clauses $T_1, \ldots, T_m, U_1, \ldots, U_n$ that define *bwReach* in *Bw*, thereby deriving a set *UnfD* of $m+n$ clauses. Then, from *UnfD* we remove all clauses whose body contains an unsatisfiable constraint and all clauses that are *subsumed* by a (distinct) constrained fact in *UnfD*.

Next we apply the *Partition* procedure and we compute a set $\{B_1, \ldots, B_h\}$ of pairwise disjoint sets of clauses, called *blocks*, such that $\mathit{UnfD} = B_1 \cup \ldots \cup B_h$.

Finally, we apply the *Generalize* procedure to each block of the partition. This generalization step is often useful because, as it has been argued in [27], it allows us to derive more efficient programs. Our *Generalize* procedure takes as input the clause D, a block B_i of the partition of *UnfD*, and the whole definition tree constructed so far. As we will indicate below, this third argument of the *Generalize* procedure allows us to express the various techniques presented in [6,13,22,27] for controlling generalization and polyvariance.

The output of the *Generalize* procedure is, for each block B_i, a definition G_i such that the constraint occurring in the body of G_i is entailed by every constraint occurring in the body of a *non-unit* clause (that is, a clause different from a constrained fact) in B_i and, hence, every non-unit clause in B_i can be folded using G_i. If all clauses in B_i are constrained facts (and thus, no folding step is required), then G_i is the special definition denoted by the symbol $\Box$. If a clause in B_i has the form $h(X) \leftarrow c(X,X') \wedge \mathit{bwReach}(X')$, then G_i has the form $\mathit{newp}(X) \leftarrow d(X) \wedge \mathit{bwReach}(X)$ and $c(X,X') \sqsubseteq d(X')$. However, we postpone the folding steps until the end of the construction of the whole tree *DefsTree*. For $i = 1, \ldots, h$, we add to *DefsTree* the arc $D \xrightarrow{B_i} G_i$.

The construction of *DefsTree* terminates when all leaf clauses of the current *DefsTree* are recurrent. In general, termination of this construction is not guaranteed and it depends on the particular *Generalize* procedure one considers. All *Generalize* procedures presented in the next section guarantee termination (see also [13,22,27]).

When the construction of *DefsTree* terminates we construct the specialized program *SpBw* by applying the *Fold* procedure which consists in: (i) collecting all clauses occurring in the blocks that label the arcs of *DefsTree*, and then (ii) folding every non-unit clause by using a definition that labels a node of *DefsTree*. Recall that, by construction, every non-unit clause occurring in a block that labels an arc of *DefsTree* can be folded by a definition that labels a node of *DefsTree*.

In the following Section, we will show how the specialization technique of Example 2 can be regarded as an instance of our generic specialization algorithm.

By using the correctness results for the unfolding, folding, and clause removal rules (see, for instance, [10]), we can prove the correctness of our generic specialization algorithm, as stated by the following theorem.

Theorem 1 (Correctness of the Specialization Algorithm). *Let programs Bw and SpBw be the input and the output programs, respectively, of the specialization algorithm that uses any given Partition, Generalize, and Fold procedures. Then* $\mathit{unsafe} \in M(\mathit{Bw})$ *iff* $\mathit{unsafe} \in M(\mathit{SpBw})$.

Input: Program Bw.
Output: Program $SpBw$ such that $unsafe \in M(Bw)$ iff $unsafe \in M(SpBw)$.

INITIALIZATION:
$DefsTree := \{\mathsf{T} \xrightarrow{\{I_1\}} D_1, \ldots, \mathsf{T} \xrightarrow{\{I_k\}} D_k\}$;
while there exists a non-recurrent definition D: $newp(X) \leftarrow c(X) \wedge bwReach(X)$ in $DefsTree$ *do*

UNFOLDING: $UnfD := \{newp(X) \leftarrow c(X) \wedge t_1(X, X') \wedge bwReach(X'),\ \ldots,$
$newp(X) \leftarrow c(X) \wedge t_m(X, X') \wedge bwReach(X'),$
$newp(X) \leftarrow c(X) \wedge u_1(X),\ \ldots,$
$newp(X) \leftarrow c(X) \wedge u_n(X)\ \}$;

CLAUSE REMOVAL:
while in $UnfD$ there exist two distinct clauses E and F such that E is a constrained fact that subsumes F or there exists a clause F whose body has a constraint which is not satisfiable *do* $UnfD := UnfD - \{F\}$ *end-while*;

DEFINITION INTRODUCTION:
$Partition(UnfD, \{B_1, \ldots, B_h\})$;
for $i = 1, \ldots, h$ *do*
$Generalize(D, B_i, DefsTree, G_i)$;
$DefsTree := DefsTree \cup \{D \xrightarrow{B_i} G_i\}$
end-for;
end-while;
FOLDING: $Fold(DefsTree, SpBw)$

Fig. 1. The generic specialization algorithm.

4 Partition, Generalize, and Fold Procedures

In this section we provide several definitions of the *Partition*, *Generalize*, and *Fold* procedures used by the generic specialization algorithm. Let us start by introducing the following notions.

First, note that the set of all conjunctions of equations on $\mathbb{D}$ can be viewed as a finite lattice whose partial order is defined by the entailment relation $\sqsubseteq$. Given the constraints $c_1, \ldots, c_n$, we define their *most specific generalization*, denoted $\gamma(c_1, \ldots, c_n)$, the conjunction of: (i) the least upper bound of the conjunctions $fd(c_1), \ldots, fd(c_n)$ of equations on $\mathbb{D}$, and (ii) the *convex hull* [6] of the constraints $re(c_1), \ldots, re(c_n)$ on $\mathbb{R}$, which is the least (w.r.t. the $\sqsubseteq$ ordering) constraint h in $\mathbb{R}$ such that $re(c_i) \sqsubseteq h$, for $i = 1, \ldots, n$. (Note that this notion of generalization is different from the one that is commonly used in logic programming.)

Note that, for $i = 1, \ldots, n$, $c_i \sqsubseteq \gamma(c_1, \ldots, c_n)$. Given a set of constraints $Cs = \{c_1, \ldots, c_n\}$, we define the equivalence relation $\simeq_{fd}$ on Cs such that, for every $c_1, c_2 \in Cs$, $c_1 \simeq_{fd} c_2$ iff $fd(c_1)$ is equivalent to $fd(c_2)$ in $\mathbb{D}$. We also define the equivalence relation $\simeq_{re}$ on Cs as the reflexive, transitive closure of the relation $\downarrow_{\mathbb{R}}$ on Cs such that, for every $c_1, c_2 \in Cs$, $c_1 \downarrow_{\mathbb{R}} c_2$ iff $re(c_1) \wedge re(c_2)$ is satisfiable in $\mathbb{R}$.

For example, let us consider an element $a \in \mathbb{D}$. Let c_1 be the constraint $X_1 > 0 \wedge X_2 = a$ and c_2 be the constraint $X_1 < 0 \wedge X_2 = a$. Then we have that

$c_1 \simeq_{fd} c_2$ on $\{c_1, c_2\}$. Now, let c_3 be the constraint $X_1 > 0 \wedge X_1 < 2$, c_4 be the constraint $X_1 > 1 \wedge X_1 < 3$, and c_5 be the constraint $X_1 > 2 \wedge X_1 < 4$. Since $c_3 \downarrow_{\mathbb{R}} c_4$ and $c_4 \downarrow_{\mathbb{R}} c_5$, we have $c_3 \simeq_{re} c_5$ on $\{c_3, c_4, c_5\}$. Note that $c_3 \not\simeq_{re} c_5$ on $\{c_3, c_5\}$ because $c_3 \wedge c_5$ is *not* satisfiable in $\mathbb{R}$.

Partition. The *Partition* procedure takes as input the following set of n (≥ 1) clauses:

$$\begin{array}{lll} \mathit{UnfD} := \{ & C_1: & \mathit{newp}(X) \leftarrow c_1(X, X') \wedge \mathit{bwReach}(X'), \\ & & \cdots \\ & C_m: & \mathit{newp}(X) \leftarrow c_m(X, X') \wedge \mathit{bwReach}(X'), \\ & C_{m+1}: & \mathit{newp}(X) \leftarrow c_{m+1}(X, X'), \\ & & \cdots \\ & C_n: & \mathit{newp}(X) \leftarrow c_n(X, X') \,\} \end{array}$$

where, for some m, with $0 \leq m \leq n$, $C_1, \ldots, C_m$ are not constrained facts, and $C_{m+1}, \ldots, C_n$ are constrained facts. The *Partition* procedure returns as output a partition $\{B_1, \ldots, B_h\}$ of *UnfD*, such that $B_h = \{C_{m+1}, \ldots, C_n\}$. The integer h and the blocks $B_1, \ldots, B_{h-1}$ are computed by using one of the following *partition operators*. For the operators *FiniteDomain*, *Constraint*, and *FDC*, the integer h to be computed is obtained as a result of the computation of the blocks B_i's.

(i) *Singleton*: $h = m+1$ and, for $1 \leq i \leq h-1$, $B_i = \{C_i\}$, which means that every non-constrained fact is in a distinct block;

(ii) *FiniteDomain*: for $1 \leq i \leq h-1$, for $j, k = 1, \ldots, m$, two clauses C_j and C_k belong to the same block B_i iff their finite domain constraints on the primed variables are equivalent, that is, iff $c_j|_{X'} \simeq_{fd} c_k|_{X'}$ on $\{c_1|_{X'}, \ldots, c_m|_{X'}\}$;

(iii) *Constraint*: for $1 \leq i \leq h-1$, for $i, j = 1, \ldots, m$, two clauses C_j and C_k belong to the same block B_i iff there exists a sequence of clauses in *UnfD* starting with C_j and ending with C_k such that for any two consecutive clauses in the sequence, the conjunction of the real constraints on the primed variables is satisfiable, that is, iff $c_j|_{X'} \simeq_{re} c_k|_{X'}$ on $\{c_1|_{X'}, \ldots, c_m|_{X'}\}$;

(iv) *FDC*: for $1 \leq i \leq h-1$, for $i, j = 1, \ldots, m$, two clauses C_j and C_k belong to the same block B_i iff they belong to the same block according to both the *FiniteDomain* and the *Constraint* partition operator, that is, iff $c_j|_{X'} \simeq_{fd} c_k|_{X'}$ and $c_j|_{X'} \simeq_{re} c_k|_{X'}$ on $\{c_1|_{X'}, \ldots, c_m|_{X'}\}$;

(v) *All*: $h = 2$ and $B_1 = \{C_1, \ldots, C_m\}$, which means that all non-constrained facts are in a single block.

Generalize. The *Generalize* procedure takes as input a definition D, a block B of clauses computed by the *Partition* procedure, and the tree *DefsTree* of definitions introduced so far, and returns a definition clause G. If B is a set of constrained facts then G is the special definition denoted by the symbol $\Box$. Otherwise, if B is the set $\{E_1, \ldots, E_k\}$ of clauses and none of which is a constrained fact, then clause G is obtained as follows.

Step 1. Let $b(X')$ denote the most specific generalization $\gamma(\mathit{con}(E_1)|_{X'}, \ldots, \mathit{con}(E_k)|_{X'})$.
if there exists a nearest ancestor A_1 of D (possibly D itself) in *DefsTree* such that A_1 is of the form: $\mathit{newq}(X') \leftarrow a_1(X') \wedge \mathit{bwReach}(X')$ (modulo

variable renaming) and $a_1(X') \simeq_{fd} con(D)$
then $b_{anc}(X') = \gamma(a_1(X'), b(X'))$ *else* $b_{anc}(X') = b(X')$;

Step 2. Let us consider a *generalization operator* $\ominus$ (see [13] and the operators *Widen* and *WidenSum* defined below).
if in *DefsTree* there exists a clause H: $newt(X') \leftarrow d(X') \wedge bwReach(X')$ (modulo variable renaming) such that $b_{anc}(X') \sqsubseteq d(X')$
then G is H
else let *newu* be a new predicate symbol
if there exists a nearest ancestor A_2 of D (possibly D itself) in *DefsTree* such that A_2 is a definition of the form:
$newr(X') \leftarrow a_2(X'), bwReach(X')$ and $a_2(X') \simeq_{fd} b_{anc}(X')$
then G is $newu(X') \leftarrow (a_2(X') \ominus b_{anc}(X')) \wedge bwReach(X')$
else G is $newu(X') \leftarrow b_{anc}(X') \wedge bwReach(X')$.

In [13] we have defined and compared several generalization operators. Among those, now we consider the following two operators which we have used in the experiments we have reported in the next section. Indeed, as indicated in [13], these two operators perform better than all other operators.

Widen. Given any two constraints c and d such that c is $a_1 \wedge \ldots \wedge a_m$, where the a_i's are atomic constraints, the operator *Widen*, denoted $\ominus_W$, returns the constraint $c \ominus_W d$ which is the conjunction of the atomic constraints of c which are entailed by d, that is, which are in the set $\{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$ (see [6] for a similar widening operator used in static analysis). Note that, in the case of our Generalize procedure, we have that $fd(d)$ is a subconjunction of $c \ominus_W d$.

WidenSum. Let us first define the thin well-quasi ordering $\precsim_S$. For any atomic constraint a on $\mathbb{R}$ of the form $q_0 + q_1 X_1 + \ldots + q_k X_k \lessdot 0$, where $\lessdot$ is either $<$ or $\leq$, we define $sumcoeff(a)$ to be $\sum_{j=0}^{k} |q_j|$. Given the two atomic constraints a_1 of the form $p_1 < 0$ and a_2 of the form $p_2 < 0$, we have that $a_1 \precsim_S a_2$ iff $sumcoeff(a_1) \leq sumcoeff(a_2)$. Similarly, if we are given the atomic constraints a_1 of the form $p_1 \leq 0$ and a_2 of the form $p_2 \leq 0$. Given any two constraints $c = a_1 \wedge \ldots \wedge a_m$ and $d = b_1 \wedge \ldots \wedge b_n$, where the a_i's and the b_i's are atomic constraints, the operator *WidenSum*, denoted $\ominus_{WS}$, returns the constraint $c \ominus_{WS} d$ which is the conjunction of the constraints in the set $\{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\} \cup \{b_k \mid b_k \text{ occurs in } re(d) \text{ and } \exists\, a_i \text{ occuring in } re(c), b_k \precsim_S a_i\}$, which is the set of atomic constraints which either occur in c and are entailed by d, or occur in d and are less than or equal to some atomic constraint in c, according to the thin well-quasi ordering $\precsim_S$. Note that, in the case of our Generalize procedure, we have that $fd(d)$ is a subconjunction of $c \ominus_{WS} d$.

Our generic Partition and Generalize procedures can be instantiated to get known specialization algorithms and abstract interpretation algorithms. In particular, (i) the technique proposed by Cousot and Halbwachs [6] can be obtained by using the operators *FiniteDomain* and *Widen*, (ii) the specialization algorithm by Peralta and Gallagher [27] can be obtained by using the operators *All* and *Widen*, and (iii) our technique presented in [13] can be obtained by using the partition operator *Singleton* together with the generalization operators *Widen* or *WidenSum*.

Fold. Let us first introduce the following definition. Given the two clauses $C: newp(X) \leftarrow c(X) \wedge bwReach(X)$ and $D: newq(X) \leftarrow d(X) \wedge bwReach(X)$, we say that C is *more general than* D, and by abuse of language, we write $D \sqsubseteq C$, iff $d(X) \sqsubseteq c(X)$. A clause C is said to be *maximally general* in a set S of clauses iff for all clauses $D \in S$, if $C \sqsubseteq D$ then $D \sqsubseteq C$. (Recall that the relation $\sqsubseteq$ is not antisymmetric.) For the *Fold* procedure we have the following two options.

Immediate Fold (*Im*, for short): (Step 1) all clauses occurring in the labels of the arcs of *DefsTree* are collected in a set F, and then (Step 2) for every non-unit clause E in F such that E occurs in the block B_i labelling an arc of the form $D \xrightarrow{B_i} D_i$, for some clause D, E is folded using D_i.

Maximally General Fold (*MG*, for short): (Step 1) is equal to that of *Immediate Fold* procedure, and (Step 2) every non-unit clause in F is folded using a maximally general clause in *DefsTree*.

Immediate Fold is simpler than *Maximally General Fold*, because it does not require any comparison between definitions in *DefsTree* to compute a maximally general one. Note also that a unique, most general definition for folding a clause may not exist, that is, there exist clauses that can be folded by using two definitions which are incomparable with respect to the $\sqsubseteq$ ordering. However, the *Maximally General Fold* procedure allows us to use a subset of the definitions introduced by the specialization algorithm, thereby reducing polyvariance and deriving specialized programs of smaller size.

As already mentioned in the previous section, the specialization technique which we have applied in Example 2 can be obtained by instantiating our generic specialization algorithm using the following operators: *Singleton* for partitioning, *Widen* for generalization, and *Immediate Fold* for folding.

5 Experimental Evaluation

We have implemented the generic specialization algorithm presented in Section 3 using MAP [25], an experimental system for transforming constraint logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `clpr` library to operate on constraints. All experiments have been performed on an Intel Core 2 Duo E7300 2.66 GHz under the Linux operating system.

We have performed the backward and forward reachability analyses of several infinite state reactive systems taken from the literature [1,2,4,8,20,28], encoding, among others, mutual exclusion protocols, cache coherence protocols, client-server systems, producer-consumer systems, array bound checking, and Reset Petri nets.

For backward reachability we have applied the method presented in Section 2. For forward reachability we have applied a variant of that method and in particular, first, (i) we have encoded the forward reachability algorithm by a constraint logic program *Fw* and we have specialized *Fw* with respect to the set of the unsafe states, thereby deriving a new program *SpFw*, and then, (ii) we have computed the least fixpoint of the immediate consequence operator S_{SpFw} (associated with program *SpFw*).

In Tables 1 and 2 we have reported the results of our verification experiments for backward reachability (that is, program *Bw*) and forward reachability (that is, program *Fw*), respectively. For each example of infinite state reactive system, we have indicated the total verification time (in milliseconds) of the non-specialized system and of the various specialized systems obtained by applying our strategy.

The symbol '∞' means that either the program specialization or the least fixpoint construction did not terminate within 200 seconds. If the time taken is less than 10 milliseconds, we have written the value '0'. Between parentheses we have also indicated the number of predicate symbols occurring in the specialized program. This number is a measure of the degree of polyvariance determined by our specialization algorithm.

In the column named *Input*, we have indicated the time taken for computing the least fixpoint of the immediate consequence operator of the input, non-specialized program (whose definition is based on program *Bw* for backward reachability, and program *Fw* for forward reachability). In the six rightmost columns, we have shown the sum of the specialization time and the time taken for computing the least fixpoint of the immediate consequence operator of the specialized programs obtained by using the following six pairs of partition operators and generalization operators: (i) $\langle$*All*, *Widen*$\rangle$, called *All_W*, (ii) $\langle$*FDC*, *Widen*$\rangle$, called *FDC_W*, (iii) $\langle$*Singleton*, *Widen*$\rangle$, called *Single_W*, (iv) $\langle$*All*, *WidenSum*$\rangle$, called *All_WS*, (v) $\langle$*FDC*, *WidenSum*$\rangle$, called *FDC_WS*, and (vi) $\langle$*Singleton*, *WidenSum*$\rangle$, called *Single_WS*. For each example the tables have two rows corresponding, respectively, to the *Immediate Fold* procedure (*Im*) and *Maximally General Fold* procedure (*MG*).

If we consider *precision*, that is, the number of successful verifications, we have that the best combinations of the partition procedure and the generalization operators are: (i) *FDC_WS* and *Single_WS* for backward reachability, each of which verified 54 properties out of 58 (in particular, 27 with *Im* and 27 with *MG*), and (ii) *Single_WS* for forward reachability, which verified 36 properties out of 58 (in particular, 18 with *Im* and 18 with *MG*).

If we compare the *Generalize* procedures we have that *WidenSum* is strictly more precise than *Widen* (up to 50%). Moreover, except for a few cases (backward reachability of CSM, forward reachability of Kanban), if a property cannot be proved by using *WidenSum* then it cannot be proved using *Widen*. *WidenSum* is usually more polyvariant than *Widen*. If we consider the verification times, they are generally favourable to *WidenSum* with respect to *Widen*, with some exceptions.

If we compare the partition operators we have that *All* is strictly less precise than the other operators: it successfully terminates in 138 cases out of 232 tests obtained by varying: (i) the given example-program, (ii) the property to be proved (either forward reachability or backward reachability), (iii) the generalization operator, and (iv) the *Fold* procedure. However, *All* is the only partition operator which allows us to verify the McCarty91 examples. By using the *Singleton* operator, the verification terminates in 158 cases out of 232, and by using the *FDC* operator, the verification successfully terminates in 159 cases out of 232. However, there are some properties (forward reachability of Peterson, InsertionSort and SelectionSort) which can only be proved using *Singleton*.

Note also that, if a property can be verified by using the *FDC* partition operator, then it can be verified by using either the operator *All* or the operator *Singleton*.

The two operators *Singleton* and *FDC* have similar polyvariance and verification times, while the operator *All* yields a specialized program with lower polyvariance and requires shorter verification times than *Singleton* and *FDC*.

If we compare the two *Fold* procedures, we have that *Maximally General Fold* for most of the examples has lower polyvariance and shorter verification times than *Immediate Fold*, while the precision of the two procedures is almost identical, except for a few cases where *Maximally General Fold* verifies the property, while *Immediate Fold* does not (backward reachability of Bakery4, Peterson and CSM).

6 Related Work and Conclusions

We have proposed a framework for controlling polyvariance during the specialization of constraint logic programs in the context of verification of infinite state reactive systems. In our framework we can combine several techniques for introducing a set of specialized predicate definitions to be used when constructing the specialized programs. In particular, we have considered new combinations of techniques introduced in the area of constraint-based program analysis and program specialization such as convex hull, widening, most specific generalization, and well-quasi orderings (see, for instance, [6,13,22,27]).

We have performed an extensive experimentation by applying our specialization framework to the reachability analysis of infinite state systems. We have considered constraint logic programs that encode both backward and forward reachability algorithms and we have shown that program specialization improves the termination of the computation of the least fixpoint needed for the analysis. However, by applying different instances of our framework, we may get different termination results and different verification times. In particular, we have provided an experimental evidence that the degree of polyvariance has an influence on the effectiveness of our specialization-based verification method.

Our experiments confirm that, on one hand, a high degree of polyvariance often corresponds to high precision of analysis (that is, high number of terminating verifications) and, on the other hand, a low degree of polyvariance often corresponds to short verification times. We have also determined a specific combination of techniques for controlling polyvariance and provides, with respect to our set of examples, a good balance between precision and verification time.

Other techniques for controlling polyvariance during the specialization of logic programs have been proposed in the literature [7,13,22,26,27]. As already

	Input	Fold	All_W	FDC_W	Single_W	All_WS	FDC_WS	Single_WS
Bakery2	60	Im	140 (5)	130 (36)	130 (36)	80 (6)	20 (23)	20 (23)
		MG	100 (3)	110 (14)	100 (14)	80 (6)	20 (15)	20 (15)
Bakery3	2710	Im	7240 (5)	3790 (144)	3870 (144)	1100 (6)	200 (77)	150 (77)
		MG	3380 (3)	2620 (64)	2190 (61)	1110 (6)	200 (60)	190 (60)
Bakery4	129900	Im	∞	112340 (535)	111540 (539)	19340 (6)	102140 (1745)	101300 (1745)
		MG	129940 (3)	37760 (292)	37010 (296)	19340 (6)	78190 (1172)	76940 (1172)
MutAst	1220	Im	4370 (6)	350 (173)	330 (173)	7850 (7)	170 (112)	150 (112)
		MG	1400 (3)	350 (59)	330 (59)	1980 (3)	190 (86)	150 (86)
Peterson N	166520	Im	∞	∞	∞	620 (9)	260 (22)	220 (22)
		MG	∞	∞	167650 (3)	650 (9)	260 (22)	230 (22)
Ticket	∞	Im	∞	30 (11)	10 (11)	∞	20 (11)	20 (11)
		MG	∞	20 (11)	20 (11)	∞	20 (11)	20 (11)
Berke-RISC	20	Im	80 (5)	70 (6)	30 (6)	70 (5)	50 (8)	40 (8)
		MG	80 (3)	70 (3)	30 (3)	70 (5)	50 (8)	30 (8)
DEC Firefly	50	Im	140 (5)	160 (7)	100 (7)	320 (7)	30 (6)	20 (6)
		MG	140 (3)	160 (3)	90 (3)	300 (5)	20 (6)	10 (6)
Futurebus+	14890	Im	16900 (6)	45240 (14)	44340 (14)	16910 (6)	2580 (19)	2410 (19)
		MG	15150 (3)	15590 (3)	14990 (3)	15140 (3)	2560 (15)	2220 (15)
Illinois Univ	70	Im	210 (5)	150 (7)	60 (7)	110 (5)	30 (6)	20 (6)
		MG	190 (3)	150 (5)	70 (5)	100 (3)	30 (6)	20 (6)
MESI	60	Im	120 (5)	50 (6)	50 (6)	90 (5)	40 (7)	20 (7)
		MG	90 (3)	60 (4)	20 (4)	90 (5)	40 (7)	30 (7)
MOESI	50	Im	220 (6)	190 (7)	130 (7)	250 (6)	90 (7)	50 (7)
		MG	200 (3)	140 (3)	90 (3)	210 (3)	90 (5)	50 (5)
Synapse N+1	10	Im	30 (4)	20 (5)	10 (5)	30 (4)	20 (5)	20 (5)
		MG	20 (3)	20 (4)	20 (4)	20 (3)	30 (4)	10 (4)
Xerox Dragon	80	Im	230 (5)	180 (7)	80 (7)	470 (7)	60 (8)	30 (8)
		MG	240 (3)	170 (5)	60 (5)	470 (5)	60 (8)	20 (8)
Barber	420	Im	290 (5)	5170 (31)	3210 (35)	750 (6)	900 (44)	300 (43)
		MG	270 (3)	3080 (6)	690 (6)	750 (6)	930 (44)	290 (43)
B-Buffer	20	Im	170 (5)	400 (11)	280 (11)	210 (6)	4490 (75)	3230 (75)
		MG	150 (3)	300 (3)	170 (3)	210 (6)	4550 (75)	3310 (75)
U-Buffer	20	Im	100 (6)	200 (12)	150 (12)	70 (6)	210 (12)	130 (12)
		MG	100 (3)	150 (4)	100 (4)	60 (3)	140 (4)	110 (4)
CSM	188110	Im	∞	∞	∞	∞	9870 (146)	6920 (154)
		MG	195700 (3)	203290 (3)	186980 (3)	∞	10310 (146)	7010 (154)
Insert Sort	40	Im	90 (7)	60 (23)	60 (23)	130 (8)	90 (28)	80 (28)
		MG	110 (7)	60 (9)	50 (9)	150 (8)	100 (14)	100 (14)
Select Sort	∞	Im	∞	∞	∞	∞	220 (35)	170 (32)
		MG	∞	∞	∞	∞	250 (19)	200 (19)
Light Control	20	Im	60 (5)	20 (9)	10 (9)	50 (5)	20 (9)	20 (9)
		MG	50 (3)	20 (7)	10 (7)	50 (3)	20 (7)	10 (7)
R-Petri Nets	∞	Im	∞	∞	∞	20 (5)	10 (5)	20 (5)
		MG	∞	∞	∞	0 (3)	0 (3)	10 (3)
GB	1750	Im	4780 (6)	3300 (10)	3300 (10)	6520 (6)	2190 (10)	2190 (10)
		MG	1870 (3)	1840 (4)	1840 (4)	1870 (3)	2070 (5)	2070 (5)
Kanban	∞	Im	∞	∞	∞	∞	8310 (162)	8170 (162)
		MG	∞	∞	∞	∞	8390 (162)	8320 (162)
McCarthy 91	∞	Im	∞	∞	∞	4130 (104)	∞	∞
		MG	∞	∞	∞	4120 (3)	∞	∞
Scheduler	∞	Im	4020 (5)	5770 (20)	5700 (20)	17530 (7)	3220 (91)	3120 (91)
		MG	2230 (3)	4730 (15)	4610 (15)	12420 (3)	3320 (83)	3220 (83)
Train	∞	Im	1710 (6)	1340 (14)	1330 (14)	3030 (8)	20250 (299)	19850 (299)
		MG	1700 (5)	970 (6)	940 (6)	3020 (7)	15730 (166)	15270 (166)
TTP	∞	Im	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Consistency	∞	Im	∞	∞	∞	350 (13)	160 (20)	160 (21)
		MG	∞	∞	∞	370 (13)	160 (20)	140 (21)
no. of successes	20	Im	19	21	21	24	27	27
		MG	21	22	23	24	27	27

Table 1. Verification Results using Backward Reachability.

	Input		*All_W*	*FDC_W*	*Single_W*	*All_WS*	*FDC_WS*	*Single_WS*
Bakery2	∞	*Im*	20 (5)	∞	∞	30 (5)	20 (20)	20 (20)
		MG	20 (5)	∞	∞	30 (5)	30 (16)	20 (16)
Bakery3	∞	*Im*	∞	∞	∞	∞	1380 (223)	1190 (240)
		MG	∞	∞	∞	∞	1450 (200)	1270 (213)
Bakery4	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
MutAst	370	*Im*	420 (4)	1790 (190)	1720 (190)	410 (4)	280 (141)	280 (141)
		MG	400 (3)	780 (51)	730 (51)	390 (3)	310 (135)	270 (135)
Peterson N	630	*Im*	∞	∞	1220 (6)	∞	∞	8000 (80)
		MG	∞	∞	730 (3)	∞	∞	8040 (80)
Ticket	50	*Im*	60 (4)	240 (30)	210 (30)	60 (4)	210 (26)	180 (26)
		MG	50 (3)	210 (11)	180 (11)	50 (3)	230 (17)	200 (17)
Berke-RISC	∞	*Im*	40 (3)	50 (3)	10 (4)	40 (3)	40 (3)	20 (4)
		MG	40 (3)	40 (3)	10 (4)	40 (3)	40 (3)	10 (4)
DEC Firefly	∞	*Im*	110 (3)	130 (3)	∞	110 (3)	100 (3)	60 (9)
		MG	100 (3)	120 (3)	∞	120 (3)	120 (3)	70 (9)
Futurebus+	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Illinois Univ	∞	*Im*	150 (3)	150 (3)	∞	140 (3)	150 (3)	70 (8)
		MG	140 (3)	140 (3)	∞	140 (3)	140 (3)	60 (8)
MESI	∞	*Im*	90 (3)	90 (3)	∞	90 (3)	90 (3)	∞
		MG	90 (3)	100 (3)	∞	90 (3)	90 (3)	∞
MOESI	∞	*Im*	130 (3)	130 (3)	∞	130 (3)	130 (3)	∞
		MG	130 (3)	130 (3)	∞	120 (3)	150 (3)	∞
Synapse N+1	∞	*Im*	10 (3)	20 (3)	0 (4)	20 (3)	20 (3)	10 (4)
		MG	20 (3)	20 (3)	0 (4)	20 (3)	20 (3)	10 (4)
Xerox Dragon	∞	*Im*	180 (3)	190 (3)	∞	190 (3)	210 (3)	80 (8)
		MG	180 (3)	190 (3)	∞	180 (3)	190 (3)	70 (8)
Barber	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
B-Buffer	∞	*Im*	∞	50 (4)	20 (4)	∞	50 (4)	20 (4)
		MG	∞	50 (4)	20 (4)	∞	50 (4)	20 (4)
U-Buffer	∞	*Im*	∞	210 (8)	70 (8)	∞	190 (8)	70 (8)
		MG	∞	230 (8)	80 (8)	∞	230 (8)	80 (8)
CSM	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Insert Sort	∞	*Im*	∞	∞	10 (14)	∞	∞	20 (14)
		MG	∞	∞	30 (14)	∞	∞	30 (14)
Select Sort	∞	*Im*	∞	∞	180 (37)	∞	∞	310 (47)
		MG	∞	∞	180 (37)	∞	∞	320 (45)
Light Control	∞	*Im*	∞	30 (18)	20 (18)	∞	30 (18)	20 (18)
		MG	∞	30 (18)	30 (18)	∞	30 (18)	20 (18)
R-Petri Nets	∞	*Im*	∞	∞	∞	0 (6)	10 (6)	0 (6)
		MG	∞	∞	∞	0 (6)	0 (6)	0 (6)
GB	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Kanban	44860	*Im*	46840 (4)	46860 (4)	56100 (13)	∞	∞	∞
		MG	45060 (3)	45210 (3)	44130 (3)	∞	∞	∞
McCarthy 91	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Scheduler	840	*Im*	910 (3)	910 (4)	1750 (32)	930 (3)	920 (4)	127370 (530)
		MG	940 (3)	910 (4)	1110 (4)	940 (3)	900 (4)	127400 (530)
Train	∞	*Im*	∞	∞	∞	∞	∞	410 (51)
		MG	∞	∞	∞	∞	∞	400 (51)
TTP	∞	*Im*	∞	∞	∞	650 (4)	1140 (15)	∞
		MG	∞	∞	∞	660 (4)	1180 (14)	∞
Consistency	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
no. of successes	5	*Im*	12	14	12	13	17	18
		MG	12	14	12	13	17	18

Table 2. Verification Results using Forward Reachability.

mentioned, the techniques presented in [13,27] can be considered as instances of our framework, while [22,26] do not consider constraints, which are of primary concern in this paper. Our framework generalizes and improves the framework of [13], by introducing partitioning and folding operators which, as shown in Section 5, increase the precision and the performance of the verification process. The *offline specialization* approach followed by [7] is based on a preliminary *binding time analysis* to decide when to unfold a call and when to introduce a new predicate definition. In the context of verification of infinite state reactive systems considered here, due to the very simple structure of the program to be specialized, deciding whether or not to unfold a call is not a relevant issue, and in our approach the binding time analysis is not performed.

As a future work we plan to continue our experiments on a larger set of infinite state reactive systems so as to enhance and better evaluate the specialization framework presented here. We also plan to extend our approach to a framework for the specialization of constraint logic programs outside the context of verification of infinite state reactive systems.

Acknowledgements

This work has been partially supported by PRIN-MIUR and by a joint project between CNR (Italy) and CNRS (France). The last author has been supported by an ERCIM grant during his stay at LORIA-INRIA. Thanks to Laurent Fribourg and John Gallagher for many stimulating conversations.

References

1. A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Proceedings of CAV 2001*, Lecture Notes in Computer Science 2102, pages 368–372. Springer, 2001.
2. G. Banda and J. P. Gallagher. Analysis of linear hybrid systems in CLP. In *Proceedings of LOPSTR 2008*, Lecture Notes in Computer Science 5438, pages 55–70. Springer, 2009.
3. G. Banda and J. P. Gallagher. Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. In *Proceedings of LPAR 2010*, Lecture Notes in Artificial Intelligence 6355, pages 27–45. Springer, 2010.
4. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.
7. S.-J. Craig and M. Leuschel. A compiler generator for constraint logic programs. In M. Broy and A. V. Zamulin, editors, *5th Ershov Memorial Conference on Perspectives of Systems Informatics, PSI 2003*, Lecture Notes in Computer Science 2890, pages 148–161. Springer, 2003.
8. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

9. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
10. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
11. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Proceedings of LOPSTR '00*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.
12. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of VCL'01*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
13. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In *Proceedings of LOPSTR 2010*, Lecture Notes in Computer Science Vol. 6564, pages 164–183. Springer, 2011.
14. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, Lecture Notes in Computer Science 3414, pages 258–273. Springer, 2005.
15. L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '99), Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 31–42. Springer-Verlag, 2000.
16. J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.
17. T. J. Hickey and D. A. Smith. Towards the partial evaluation of CLP languages. In *Proceedings of the 1991 ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, New Haven, CT, USA*, SIGPLAN Notices, 26, 9, pages 43–51. ACM Press, 1991.
18. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.
19. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
20. LASH. homepage: http://www.montefiore.ulg.ac.be/∼boigelot/research/lash.
21. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.
22. 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.
23. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In *Proceedings of LOPSTR '99*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 2000.
24. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
25. MAP. The MAP transformation system.
Available from http://www.iasi.cnr.it/∼proietti/system.html, 1995–2010.

26. C. Ochoa, G. Puebla, and M. V. Hermenegildo. Removing superfluous versions in polyvariant specialization of prolog programs. In *Proceedings of LOPSTR '05*, Lecture Notes in Computer Science 3961, pages 80–97. Springer, 2006.
27. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *Proceedings of LOPSTR '02*, Lecture Notes in Computer Science 2664, pages 90–108. Springer, 2003.
28. 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.