Controlling Polyvariance for Specialization-Based Verification

Fabio Fioravanti\textsuperscript{1}, Alberto Pettorossi\textsuperscript{2}, Maurizio Proietti\textsuperscript{3}, and Valerio Senni\textsuperscript{2,4}

\textsuperscript{1} Dipartimento di Scienze, University ‘G. D’Annunzio’, Pescara, Italy
fioravanti@sci.unich.it
\textsuperscript{2} DISP, University of Rome Tor Vergata, Rome, Italy
\{pettorossi,senni\}@disp.uniroma2.it
\textsuperscript{3} CNR-IASI, Rome, Italy
maurizio.proietti@iasi.cnr.it
\textsuperscript{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 \( <a_1, \ldots, a_n> \) 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 \( <X_1, \ldots, X_n> \) 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 inequalities
on \( \mathbb{R} \). An equation on \( \mathbb{R} \) is considered as a conjunction of two inequalities. Given
a constraint \( c \), every equation in \( fd(c) \) and every linear inequality in \( re(c) \) is said
to be an atomic constraint.

The set \( I \) of the initial states is represented by a disjunction \( \text{init}_1(X) \vee \ldots \vee \text{init}_k(X) \) 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 \( <X'_1, \ldots, X'_n> \) 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 \( H \leftarrow c \land \land G \), by \( \text{con}(C) \) 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 \( H \leftarrow d \land \land G \), 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 \( u_1(X) \vee \ldots \vee u_m(X) \) 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 $Bw$:

$$
\begin{align*}
I_1: & \text{unsafe} \leftarrow \text{init}_1(X) \land \text{bwReach}(X) \\
I_k: & \text{unsafe} \leftarrow \text{init}_k(X) \land \text{bwReach}(X) \\
T_1: & \text{bwReach}(X) \leftarrow t_1(X, X') \land \text{bwReach}(X') \\
T_m: & \text{bwReach}(X) \leftarrow t_m(X, X') \land \text{bwReach}(X') \\
U_1: & \text{bwReach}(X) \leftarrow u_1(X) \\
U_n: & \text{bwReach}(X) \leftarrow u_n(X)
\end{align*}
$$

We have that: (i) $\text{bwReach}(X)$ holds iff an unsafe state can be reached from the state $X$ in zero or more applications of the transition relation, and (ii) $\text{unsafe}$ 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 $D$ 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 $\text{unsafe} \notin M(Bw)$.

**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 $X_1 \geq 1 \land X_2 = 0$ 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 $X'_1 = X_1 + X_2 \land X'_2 = X_2 + 1$ 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 $\text{unsafe}$ is defined by the following CLP program $Bw1$:

1. $\text{unsafe} \leftarrow X_1 \geq 1 \land X_2 = 0 \land \text{bwReach}(X_1, X_2)$
2. $\text{bwReach}(X_1, X_2) \leftarrow X'_1 = X_1 + X_2 \land X'_2 = X_2 + 1 \land \text{bwReach}(X'_1, X'_2)$
3. $\text{bwReach}(X_1, X_2) \leftarrow X_2 > X_1$

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 $A\theta$ such that there exists a constrained fact $A \leftarrow c$ in $S_{Bw} \uparrow \omega$ and the constraint $c\theta$ is satisfiable. $BR$ is the set of all states $s$ such that there exists a constrained fact of the form $bwReach(X) \leftarrow c(X)$ 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) $unsafe \in M(Bw)$ if and only if $unsafe \in M(SpBw)$, 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 $X_1 \geq 1 \land X_2 = 0$. (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 $X_1 \geq 1 \land X_2 = 0$. 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. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 = 0 \land bwReach(X_1, X_2)$

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

5. $unsafe \leftarrow X_1 \geq 1 \land X_2 = 0 \land new1(X_1, X_2)$

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 $bwReach(X_1, X_2)$ by the bodies of the clauses 2 and 3 defining $bwReach(X_1, X_2)$ in $Bw1$, and we derive:

6. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 = 0 \land X_1' = X_1 \land X_2' = 1 \land bwReach(X_1', X_2')$
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 $X_1 \geq 1 \land X_2 = 0 \land X_1' = X_1 \land X_2' = 1$ projected w.r.t. the variables $(X_1', X_2')$:

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

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. $(X_1', X_2')$, then we will introduce, in fact, an infinite sequence of new predicate definitions of the form:

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

where $k$ gets the values 1, 2, . . . . 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. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 \geq 0 \land bwReach(X_1, X_2)$

We fold clause 6 using clause 8 and we get:

9. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 = 0 \land X_1' = X_1 \land X_2' = 1 \land new2(X_1', X_2')$

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. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 \geq 0 \land X_1' = X_1 + X_2 \land X_2' = X_2 + 1 \land new2(X_1', X_2')$

11. $new2(X_1, X_2) \leftarrow X_1 \geq 1 \land X_2 > X_1$

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

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 $newp(X) \leftarrow d(X) \land bwReach(X)$, 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 $\{T \xrightarrow{\{I_1\}} D_1, \ldots, T \xrightarrow{\{I_k\}} D_k\}$, 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 $new_j(X) \leftarrow init_j(X) \land bwReach(X)$, 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 $UnfD$. 

7
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 \( \text{UnfD} = B_1 \cup \ldots \cup B_h \).

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 \( \text{UnfD} \), 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 \( h(X) \leftarrow c(X,X') \land \text{bwReach}(X') \), then \( G_i \) has the form \( \text{newp}(X) \leftarrow d(X) \land \text{bwReach}(X) \) and \( c(X,X') \subseteq d(X') \). However, we postpone the folding steps until the end of the construction of the whole tree \( \text{DefsTree} \).

For \( i = 1, \ldots, h \), we add to \( \text{DefsTree} \) the arc \( D \xrightarrow{B_i} G_i \).

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

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 \( \text{SpBw} \) be the input and the output programs, respectively, of the specialization algorithm that uses any given Partition, Generalize, and Fold procedures. Then \( \text{unsafe} \in M(Bw) \iff \text{unsafe} \in M(\text{SpBw}) \).
Input: Program $Bw$.
Output: Program $SpBw$ such that $\text{unsafe} \in M(Bw)$ iff $\text{unsafe} \in M(SpBw)$.

Initialization:
$$\text{DefsTree} := \{ \{ t_1 \} D_1, \ldots, \{ t_k \} D_k \};$$
while there exists a non-recurrent definition $D$: $\text{newp}(X) \leftarrow c(X) \land \text{bwReach}(X)$ in $\text{DefsTree}$ do

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

Clause Removal:
while in $\text{UnfD}$ 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 $\text{UnfD} := \text{UnfD} - \{ F \}$ end-while;

Definition Introduction:

Partition($\text{UnfD}$, $\{ B_1, \ldots, B_h \}$);

for $i = 1, \ldots, h$ do

Generalize($D, B_i, \text{DefsTree}, G_i$);

$\text{DefsTree} := \text{DefsTree} \cup \{ D \xrightarrow{B_i} G_i \}$
end-for;

end-while;

Folding: $\text{Fold}(\text{DefsTree}, \text{SpBw})$

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 $\text{fd}(c_1), \ldots, \text{fd}(c_n)$ 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 $\simeq_{\text{fd}}$ on $Cs$ such that, for every $c_1, c_2 \in Cs$, $c_1 \simeq_{\text{fd}} c_2$ iff $\text{fd}(c_1)$ is equivalent to $\text{fd}(c_2)$ in $\mathbb{D}$. We also define the equivalence relation $\simeq_{re}$ on $Cs$ as the reflexive, transitive closure of the relation $\sqsubseteq$ on $Cs$ such that, for every $c_1, c_2 \in Cs$, $c_1 \sqsubseteq c_2$ iff $re(c_1) \land re(c_2)$ is satisfiable in $\mathbb{R}$.

For example, let us consider an element $a \in \mathbb{D}$. Let $c_1$ be the constraint $X_1 > 0 \land X_2 = a$ and $c_2$ be the constraint $X_1 < 0 \land X_2 = a$. Then we have that
$c_1 \simeq_{fd} c_2$ on \{c_1, c_2\}. Now, let $c_3$ be the constraint $X_1 > 0 \land X_1 < 2$, $c_4$ be the constraint $X_1 > 1 \land X_1 < 3$, and $c_5$ be the constraint $X_1 > 2 \land X_1 < 4$. Since $c_3 \not\simeq_R c_4$ and $c_4 \not\simeq_R c_5$, we have $c_3 \not\simeq_R c_5$ on \{c_3, c_4, c_5\}. Note that $c_3 \not\simeq_R c_5$ on \{c_3, c_5\} because $c_3 \land c_5$ is not satisfiable in $R$.

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

$$UnfD := \{C_1 : \text{newp}(X) \leftarrow c_2(X, X') \land bwReach(X'),\
\quad \ldots\quad\
\quad C_m : \text{newp}(X) \leftarrow c_m(X, X') \land bwReach(X'),\
\quad C_{m+1} : \text{newp}(X) \leftarrow c_{m+1}(X, X'),\
\quad \ldots\quad\
\quad C_n : \text{newp}(X) \leftarrow c_n(X, X') \}$$

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 $c_j|X' \simeq_{fd} c_k|X_1$ and $c_j|X' \simeq_{re} c_k|X_2$ 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 $\gamma(con(E_1)|X’, \ldots, con(E_k)|X')$.

if there exists a nearest ancestor $A_1$ of $D$ (possibly $D$ itself) in DefsTree such that $A_1$ is of the form: $\text{newp}(X') \leftarrow a_1(X') \land bwReach(X')$ (modulo
variable renaming) and \( a_1(X') \succeq_{fd} \text{con}(D) \)
then \( b_{\text{anc}}(X') = \gamma(a_1(X'), b(X')) \) else \( b_{\text{anc}}(X') = b(X') \);

**Step 2.** Let us consider a generalization operator \( \odot \) (see [13] and the operators Widen and WidenSum defined below).
if inDefsTree there exists a clause \( H \): \( \text{new}(X') \leftarrow d(X') \land \text{bwReach}(X') \)
(modulo variable renaming) such that \( b_{\text{anc}}(X') \sqsubseteq d(X') \)
then \( G = H \)
else let \text{new}u be a new predicate symbol
if there exists a nearest ancestor \( A_2 \) of \( D \) (possibly \( D \) itself) inDefsTree
such that \( A_2 \) is a definition of the form:
\( \text{new}(X') \leftarrow a_2(X'), \text{bwReach}(X') \) and \( a_2(X') \succeq_{fd} b_{\text{anc}}(X') \)
then \( G = \text{new}(X') \leftarrow (a_2(X') \sqcup b_{\text{anc}}(X')) \land \text{bwReach}(X') \)
else \( G = \text{new}(X') \leftarrow b_{\text{anc}}(X') \land \text{bwReach}(X') \).

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 \( c = a_1 \land \ldots \land a_m \), where the \( a_i \)'s are atomic constraints, the operator Widen, denoted \( \odot_W \), returns the constraint \( c \odot_W d \) which is the conjunction of the atomic constraints of \( c \) which are entailed by \( d \), that is, which are in the set \( \{a_i \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h \} \)
(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 \( c \odot_W d \).

**WidenSum.** Let us first define the thin well-quasi ordering \( \preceq_S \). For any atomic constraint \( a \) on \( \mathbb{R} \) of the form \( q_0 + q_1 X_1 + \ldots + q_k X_k < 0 \), where \( < \) is either \( < \) or \( \leq \), we define \( \text{sumcoeff}(a) \) 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 \( a_1 \preceq_S a_2 \) iff \( \text{sumcoeff}(a_1) \leq \text{sumcoeff}(a_2) \). 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 \( c = a_1 \land \ldots \land a_m \) and \( d = b_1 \land \ldots \land b_n \), where the \( a_i \)'s and the \( b_i \)'s are atomic constraints, the operator WidenSum, denoted \( \odot_{WS} \), returns the constraint \( c \odot_{WS} d \) which is the conjunction of the constraints in the set \( \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h \} \cup \{b_k \mid b_k \text{ occurs in } rv(d) \text{ and } \exists a_i \text{ occurring in } rv(c), b_k \not\preceq_S a_i \} \), 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 \( \preceq_S \). Note that, in the case of our Generalize procedure, we have that \( fd(d) \) is a subconjunction of \( c \odot_{WS} d \).

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 $\text{Singleton}$ together with the generalization operators $\text{Widen}$ or $\text{WidenSum}$.

**Fold.** Let us first introduce the following definition. Given the two clauses $C: \text{newp}(X) \leftarrow c(X) \land \text{bwReach}(X)$ and $D: \text{newq}(X) \leftarrow d(X) \land \text{bwReach}(X)$, 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 ($\text{Im}$, for short): (Step 1) all clauses occurring in the labels of the arcs of $\text{DefsTree}$ 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 $D_i \xrightarrow{B_i} D_j$, for some clause $D, E$ is folded using $D_j$.

Maximally General Fold ($\text{MG}$, 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 $\text{DefsTree}$.

Immediate Fold is simpler than Maximally General Fold, because it does not require any comparison between definitions in $\text{DefsTree}$ 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: $\text{Singleton}$ for partitioning, $\text{Widen}$ 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 c1pr 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 ‘$\infty$’ 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 AllW, (ii) $\langle$FDC, Widen$\rangle$, called FDCW, (iii) $\langle$Singleton, Widen$\rangle$, called SingleW, (iv) $\langle$All, WidenSum$\rangle$, called AllWS, (v) $\langle$FDC, WidenSum$\rangle$, called FDCWS, and (vi) $\langle$Singleton, WidenSum$\rangle$, called SingleWS. 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) FDCWS and SingleWS for backward reachability, each of which verified 54 properties out of 58 (in particular, 27 with $Im$ and 27 with $MG$), and (ii) SingleWS 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
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bakery2</td>
<td>69</td>
<td>Im</td>
<td>140 (9)</td>
<td>130 (9)</td>
<td>130 (9)</td>
<td>80 (6)</td>
<td>20 (23)</td>
<td>20 (23)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bakery3</td>
<td>2710</td>
<td>Im</td>
<td>7290 (6)</td>
<td>3790 (14)</td>
<td>3870 (14)</td>
<td>1100 (6)</td>
<td>200 (77)</td>
<td>150 (77)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bakery4</td>
<td>129900</td>
<td>Im</td>
<td>∞</td>
<td>112340 (55)</td>
<td>111540 (55)</td>
<td>19340 (6)</td>
<td>101240 (1745)</td>
<td>101300 (1745)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NutAst</td>
<td>1220</td>
<td>4370 (6)</td>
<td>350 (173)</td>
<td>330 (173)</td>
<td>7850 (7)</td>
<td>170 (112)</td>
<td>150 (112)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peterson N</td>
<td>116520</td>
<td>Im</td>
<td>∞</td>
<td>420 (3)</td>
<td>50 (3)</td>
<td>190 (9)</td>
<td>200 (3)</td>
<td>90 (3)</td>
<td>150 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ticket</td>
<td></td>
<td>Im</td>
<td>∞</td>
<td>30 (11)</td>
<td>10 (11)</td>
<td>∞</td>
<td>20 (11)</td>
<td>20 (11)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge-HISC</td>
<td>20</td>
<td>Im</td>
<td>80 (3)</td>
<td>70 (3)</td>
<td>30 (3)</td>
<td>70 (8)</td>
<td>90 (5)</td>
<td>40 (8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC Firefly</td>
<td>50</td>
<td>140 (5)</td>
<td>160 (7)</td>
<td>100 (7)</td>
<td>320 (7)</td>
<td>30 (6)</td>
<td>20 (6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futurebus</td>
<td>14890</td>
<td>Im</td>
<td>16900 (6)</td>
<td>42340 (14)</td>
<td>44340 (14)</td>
<td>16910 (6)</td>
<td>2580 (19)</td>
<td>2410 (19)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Illinois Univ</td>
<td>70</td>
<td>210 (5)</td>
<td>150 (7)</td>
<td>60 (7)</td>
<td>110 (6)</td>
<td>30 (6)</td>
<td>20 (6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE3I</td>
<td>60</td>
<td>120 (5)</td>
<td>50 (6)</td>
<td>50 (6)</td>
<td>90 (5)</td>
<td>40 (7)</td>
<td>20 (7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE3SI</td>
<td>50</td>
<td>220 (5)</td>
<td>190 (7)</td>
<td>130 (7)</td>
<td>250 (6)</td>
<td>90 (7)</td>
<td>50 (7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synapse #1</td>
<td>10</td>
<td>Im</td>
<td>30 (4)</td>
<td>20 (5)</td>
<td>10 (5)</td>
<td>30 (4)</td>
<td>20 (5)</td>
<td>50 (5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xerox Dragon</td>
<td>50</td>
<td>230 (5)</td>
<td>180 (7)</td>
<td>80 (7)</td>
<td>470 (7)</td>
<td>60 (8)</td>
<td>30 (8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barber</td>
<td>420</td>
<td>290 (6)</td>
<td>5170 (31)</td>
<td>3210 (35)</td>
<td>760 (6)</td>
<td>900 (44)</td>
<td>300 (44)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-Buffer</td>
<td>25</td>
<td>170 (5)</td>
<td>400 (11)</td>
<td>280 (11)</td>
<td>210 (6)</td>
<td>4490 (75)</td>
<td>3230 (75)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P-Buffer</td>
<td>20</td>
<td>100 (9)</td>
<td>200 (12)</td>
<td>150 (12)</td>
<td>70 (6)</td>
<td>210 (12)</td>
<td>130 (12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSM</td>
<td>188110</td>
<td>Im</td>
<td>195700 (3)</td>
<td>203290 (3)</td>
<td>186980 (3)</td>
<td>∞</td>
<td>9870 (146)</td>
<td>6920 (154)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert Sort</td>
<td>40</td>
<td>Im</td>
<td>90 (7)</td>
<td>60 (23)</td>
<td>60 (23)</td>
<td>130 (8)</td>
<td>90 (28)</td>
<td>80 (28)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select Sort</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>220 (35)</td>
<td>170 (32)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Light Control</td>
<td>20</td>
<td>50 (5)</td>
<td>20 (9)</td>
<td>10 (9)</td>
<td>50 (5)</td>
<td>20 (9)</td>
<td>20 (9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-Petri Nets</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>20 (5)</td>
<td>10 (5)</td>
<td>20 (5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB</td>
<td>1750</td>
<td>4780 (8)</td>
<td>3300 (10)</td>
<td>3300 (10)</td>
<td>6520 (6)</td>
<td>2190 (10)</td>
<td>2190 (10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nanban</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>8310 (162)</td>
<td>8170 (162)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>McCarthy 91</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>4130 (104)</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td>∞</td>
<td>Im</td>
<td>4020 (5)</td>
<td>5770 (20)</td>
<td>5700 (20)</td>
<td>17830 (7)</td>
<td>3220 (91)</td>
<td>3120 (91)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Train</td>
<td>∞</td>
<td>Im</td>
<td>1710 (6)</td>
<td>1340 (14)</td>
<td>1330 (14)</td>
<td>3030 (8)</td>
<td>20250 (299)</td>
<td>19850 (299)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFP</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>50 (3)</td>
<td>0 (3)</td>
<td>10 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistency</td>
<td>∞</td>
<td>Im</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>350 (13)</td>
<td>160 (20)</td>
<td>160 (21)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no. of successes</td>
<td>20</td>
<td>Im</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GM</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>27</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Verification Results using Backward Reachability.
<table>
<thead>
<tr>
<th>Input</th>
<th>All_W</th>
<th>FDC_W</th>
<th>Single_W</th>
<th>All_WS</th>
<th>FDC_WS</th>
<th>Single_WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bakery2</td>
<td>∞</td>
<td>20 (5)</td>
<td>∞</td>
<td>30 (5)</td>
<td>20 (20)</td>
<td>20 (20)</td>
</tr>
<tr>
<td>Bakery3</td>
<td>∞</td>
<td>20 (5)</td>
<td>∞</td>
<td>30 (5)</td>
<td>20 (16)</td>
<td>20 (16)</td>
</tr>
<tr>
<td>Bakery4</td>
<td>∞</td>
<td>20 (5)</td>
<td>∞</td>
<td>30 (16)</td>
<td>20 (16)</td>
<td>20 (16)</td>
</tr>
<tr>
<td>NutAst</td>
<td>370</td>
<td>420 (4)</td>
<td>1790 (50)</td>
<td>1720 (190)</td>
<td>410 (4)</td>
<td>280 (141)</td>
</tr>
<tr>
<td>Peterson</td>
<td>830</td>
<td>∞</td>
<td>1220 (6)</td>
<td>∞</td>
<td>8000 (80)</td>
<td>8000 (80)</td>
</tr>
<tr>
<td>Ticket</td>
<td>50</td>
<td>60 (4)</td>
<td>200 (30)</td>
<td>210 (30)</td>
<td>200 (26)</td>
<td>180 (26)</td>
</tr>
<tr>
<td>Heres-RISC</td>
<td>∞</td>
<td>50 (3)</td>
<td>200 (3)</td>
<td>10 (4)</td>
<td>40 (3)</td>
<td>20 (3)</td>
</tr>
<tr>
<td>SEC Firefly</td>
<td>∞</td>
<td>110 (3)</td>
<td>130 (3)</td>
<td>110 (3)</td>
<td>100 (3)</td>
<td>60 (9)</td>
</tr>
<tr>
<td>Futurebus+</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Illinois Univ</td>
<td>∞</td>
<td>150 (3)</td>
<td>150 (3)</td>
<td>150 (3)</td>
<td>70 (8)</td>
<td>70 (8)</td>
</tr>
<tr>
<td>RESI</td>
<td>∞</td>
<td>90 (3)</td>
<td>90 (3)</td>
<td>90 (3)</td>
<td>60 (8)</td>
<td>60 (8)</td>
</tr>
<tr>
<td>NRESI</td>
<td>∞</td>
<td>130 (3)</td>
<td>130 (3)</td>
<td>130 (3)</td>
<td>130 (3)</td>
<td>130 (3)</td>
</tr>
<tr>
<td>Synapse N+1</td>
<td>∞</td>
<td>10 (3)</td>
<td>20 (3)</td>
<td>0 (4)</td>
<td>20 (3)</td>
<td>10 (4)</td>
</tr>
<tr>
<td>Xerox Dragon</td>
<td>∞</td>
<td>180 (3)</td>
<td>190 (3)</td>
<td>190 (3)</td>
<td>80 (8)</td>
<td>80 (8)</td>
</tr>
<tr>
<td>Barber</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>U-Buffer</td>
<td>∞</td>
<td>50 (4)</td>
<td>20 (4)</td>
<td>30 (4)</td>
<td>20 (4)</td>
<td>20 (4)</td>
</tr>
<tr>
<td>Q-Buffer</td>
<td>∞</td>
<td>210 (8)</td>
<td>70 (8)</td>
<td>190 (8)</td>
<td>70 (8)</td>
<td>70 (8)</td>
</tr>
<tr>
<td>CBM</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Insert Sort</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Select Sort</td>
<td>∞</td>
<td>∞</td>
<td>180 (37)</td>
<td>∞</td>
<td>310 (47)</td>
<td>310 (47)</td>
</tr>
<tr>
<td>Light Control</td>
<td>∞</td>
<td>30 (18)</td>
<td>20 (18)</td>
<td>30 (18)</td>
<td>20 (18)</td>
<td>20 (18)</td>
</tr>
<tr>
<td>R-Petri Nets</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>10 (6)</td>
<td>10 (6)</td>
</tr>
<tr>
<td>GB</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Kanban</td>
<td>44860</td>
<td>46840 (4)</td>
<td>46860 (4)</td>
<td>51600 (13)</td>
<td>127370 (530)</td>
<td>127370 (530)</td>
</tr>
<tr>
<td>McCarthy 91</td>
<td>∞</td>
<td>45660 (3)</td>
<td>45210 (3)</td>
<td>44130 (3)</td>
<td>127400 (530)</td>
<td>127400 (530)</td>
</tr>
<tr>
<td>Scheduler</td>
<td>840</td>
<td>910 (3)</td>
<td>910 (4)</td>
<td>1750 (32)</td>
<td>930 (3)</td>
<td>920 (4)</td>
</tr>
<tr>
<td>Train</td>
<td>∞</td>
<td>940 (3)</td>
<td>910 (4)</td>
<td>1110 (4)</td>
<td>940 (3)</td>
<td>900 (4)</td>
</tr>
<tr>
<td>FTP</td>
<td>∞</td>
<td>∞</td>
<td>660 (4)</td>
<td>1140 (18)</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Consistency</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

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


