Fundamenta Informaticae ??? (2009) 1–19
IOS Press

Controlling Polyvariance for Specialization-based Verification

Fabio Fioravanti
University of Chieti-Pescara
Viale Pindaro 42, 65127 Pescara, Italy
fioravanti@sci.unich.it

Alberto Pettorossi
University of Rome Tor Vergata
Via del Politecnico 1, 00133 Rome, Italy
pettorossi@disp.uniroma2.it

Maurizio Proietti
IASI-CNR
Viale Manzoni 30, 00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Valerio Senni
IMT Institute for Advanced Studies Lucca
Piazza San Ponziano 6, 55100 Lucca, Italy
valerio.senni@imtlucca.it

Abstract. Program specialization has been proposed as a means of improving constraint-based analysis of infinite state reactive systems. In particular, safety properties can be specified by constraint logic programs encoding (backward or forward) reachability algorithms. These programs are then 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). By using the specialized reachability programs, we can considerably increase the number of successful verifications. An important feature of specialization algorithms is the so called *polyvariance*, that is, the number of specialized variants of the same predicate that are introduced by specialization. Depending on this feature, the specialization time, the size of the specialized program, and the number of successful verifications may vary. We present a specialization framework which is more general than previous proposals and provides control on polyvariance. We demonstrate, through experiments on several infinite state reactive systems, that by a careful choice of the degree of polyvariance we can design specialization-based verification procedures that are both efficient and precise.

Address for correspondence: Valerio Senni, IMT Institute for Advanced Studies Lucca, Piazza San Ponziano 6, 55100 Lucca, Italy, valerio.senni@imtlucca.it

Keywords: Program specialization, constraint logic programming, polyvariance, generalization, verification of infinite state reactive systems, unfold/fold transformation.

1. Introduction

Program specialization, also known as partial evaluation, 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 [23]. Program specialization techniques have been proposed for several programming languages and, in particular, for (constraint) logic languages (see [8, 14, 20, 21, 25, 26, 28, 31]).

Specialization may be *polyvariant*, in the sense that it may derive, starting from a single procedure, several specialized versions of that procedure. In the case of (constraint) logic programming, program specialization may introduce several new predicates corresponding to specialized versions of the same predicate in the original program. The application of specialized procedures to specific input values often results in more efficient computations. However, if the number of new predicate definitions, and hence the size of the specialized program, is overly large, we may lose the advantages of specialization.

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 [26, 30], 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 [15, 17, 16, 27].

Applying model checking [5] to systems with an infinite number of states is a challenging task, because exhaustive state exploration is impossible and, even for restricted classes, simple properties such as *safety* (or *reachability*) properties are undecidable (see [12] for a survey of relevant results).

Some authors have advocated the use of symbolic approaches using *constraints* to represent infinite sets of states and constraint logic programs to encode temporal properties (see, for instance, [9, 19]). In these approaches, the verification of temporal properties is performed by computing the least (or the greatest, depending on the properties) 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 to improve termination and they are based on *abstract interpretation* [2, 3, 7, 9] and *program specialization* [15, 16, 17, 27].

The techniques based on abstract interpretation compute program *invariants*, that is, constraints that hold in the program model, which are sometimes sufficient to prove that the safety property of interest holds. However, if the safety property is not a consequence of the computed invariants, one can conclude nothing about the safety of the system, even if the system is indeed safe. In this case one can generate *spurious* counterexamples, that is, abstract computation paths leading to unsafe states, which do not correspond to any concrete computation path. To alleviate this problem, the *counterexample guided abstraction refinement* (CEGAR) technique has been proposed to automatically strengthen the computed invariants so that spurious counterexamples are avoided [6]. However, new spurious counterexamples might be generated for the strengthened invariants and the refinement process of CEGAR is not guaranteed to terminate, as the strengthening process may be repeated an unbounded number of times.

Specialization-based techniques aim at improving termination of the verification process by transforming the program that encodes both the infinite state system and the safety property of interest. Pro-

gram specialization transforms the given program by taking into account the property to be proved and the initial states of the system. Similarly to abstract interpretation, also specialization computes invariants, although in an implicit way. These invariants are used to transform the program so that the construction of the model of the transformed program terminates more often than that of the original program.

This paper extends previous work in [15, 17, 16], by proposing a more general specialization framework and a parameterized generalization strategy (that can also be instantiated to previously proposed generalization strategies), that allows us to control the degree of polyvariance of specialized programs. In order to simplify the presentation, in this paper we only consider reachability properties, but the extension to full CTL [5] is straightforward and requires only to encode the CTL temporal operators, as shown in [17]. We show that the control of polyvariance plays a relevant role for the development of effective verification techniques based on program specialization: the specialization time, the size of the specialized program, and the precision of the verification may vary, depending on the degree of polyvariance introduced by different strategies. Through several examples of infinite state reactive systems, we compare the effectiveness of various polyvariance control strategies and we show that polyvariance control is useful to increment the precision of the analysis while reducing the overall verification time.

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 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.

2. Specialization-Based Reachability Analysis of Infinite State Reactive Systems

We begin by defining a language of constraints, which will be used for the symbolic representation of infinite state reactive systems.

Let $\mathbb{D}$ be a finite set and $\mathbb{R}$ be the set of the real numbers. An *atomic constraint* is an equality on $\mathbb{D}$ or a linear inequality on $\mathbb{R}$. A *constraint* c is a finite conjunction of atomic constraints and it can also be denoted by $c(V)$, to indicate that the variables occurring in c are among the ones in the tuple V of variables. By $fd(c)$ we denote the conjunction of the equalities on $\mathbb{D}$ occurring in c and by $re(c)$ we denote the conjunction of the linear inequalities on $\mathbb{R}$ occurring in c. Given a constraint c and a tuple U of variables, the *projection* $c|_U$ is the constraint d such that: (i) the variables of d are among the variables in U, and (ii) $\mathbb{D} \cup \mathbb{R} \models \forall(d \leftrightarrow \exists Z\, c)$ where Z is the tuple of the variables occurring in c and not in U. The set of constraints we consider in this paper is closed under projection. We say that a constraint c *entails* a constraint d, written $c \sqsubseteq d$, iff $\mathbb{D} \cup \mathbb{R} \models \forall(c \rightarrow d)$. We say that c is *equivalent* to d, written $c \equiv d$, iff $c \sqsubseteq d$ and $d \sqsubseteq c$.

We extend the language of constraints by defining constraint logic programs, which will be used to define properties of infinite state reactive systems.

An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, for $n \geq 0$, where p is a predicate symbol, different from equality and inequality, and $t_1, \ldots, t_n$ are terms. A *clause* C is a formula of the form $H \leftarrow c \wedge G$, where H is an atom, c is a constraint and, G is a goal, that is, a conjunction of atoms. A *constraint logic program* is a set of clauses. By $ct(C)$ we denote the constraint c in the clause C. If G is the empty conjunction, then C is said to be a *constrained fact*, otherwise C is said to be a *proper clause*. We say

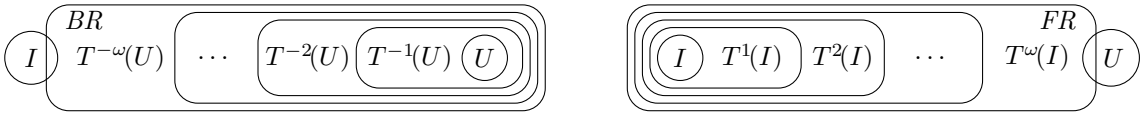

Figure 1. Backward (BR) and Forward (FR) strategies illustrated.

that a constrained fact $H \leftarrow c$ *subsumes* a clause $H \leftarrow d \wedge G$ iff $d \sqsubseteq c$. The semantics of a constraint logic program *P* is given by its *least model*, denoted $M(P)$, that is, the set of ground atoms derived by using: (i) the theory of equalities over $\mathbb{D}$ and the theory of linear inequalities over $\mathbb{R}$ for the evaluation of the constraints, and (ii) the usual least model construction (see [22] for more details).

An infinite state reactive system is represented by using constraints, as follows. A *state* is an tuple $\langle a_1, \ldots, a_p \rangle$, where each a_i belongs either to $\mathbb{D}$ or to $\mathbb{R}$. By X we denote an tuple $\langle X_1, \ldots, X_q \rangle$ of variables where each X_i ranges over either $\mathbb{D}$ or $\mathbb{R}$. The set I of the *initial states* is represented by a disjunction $\mathit{init}_1(X) \vee \ldots \vee \mathit{init}_k(X)$ of constraints. The *transition relation* T is represented by a disjunction $t_1(X, X') \vee \ldots \vee t_m(X, X')$ of constraints, where X' is the tuple $\langle X'_1, \ldots, X'_n \rangle$ of primed variables. In this paper we focus on the verification of a class of properties of infinite state reactive systems called *safety* properties. Let the set U of the unsafe states of a system be represented by a disjunction $u_1(X) \vee \ldots \vee u_n(X)$ of constraints. A safety property holds iff *no unsafe state can be reached from an initial state of the system*.

Typically, one can verify a safety property by using one of the following two strategies:
(i) the *Backward Strategy*, which consists in computing the set *BR* of states from which it is possible to reach an *unsafe* state, and then checking whether or not *BR* has an empty intersection with the set I of the initial states;
(ii) the *Forward Strategy*, which consists in computing the set *FR* of states reachable from an initial state, and then checking whether or not *FR* has an empty intersection with the set U of the unsafe states.
The Backward and Forward Strategies are illustrated in Figure 1. A number of variants of these two strategies have been proposed and implemented in various automatic verification tools [1, 4, 18, 24, 32].

Those two strategies can be easily encoded into constraint logic programming. In particular, we can encode the backward reachability strategy by means of the following constraint logic program *Bw*:

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

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 inverse transition relation, and (ii) *unsafe* holds iff there exists an initial state of the system from which an unsafe state can be reached. The system is *safe* iff $\mathit{unsafe} \notin M(\mathit{Bw})$.

Example 2.1. 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 constraint logic program *Bw*1:

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 $S_{Bw}\uparrow\omega$ [22]. The operator S_{Bw} is analogous to the usual immediate consequence operator associated with a logic program, except that it constructs a set of constrained facts, instead of a set of ground atoms. In particular, $M(Bw)$ 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* belongs to $M(Bw)$ 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 in the initial states. This may lead to a failure of the verification process 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 constraints holding in the unsafe states when computing *FR*.

In this paper we present a method, based on the program specialization technique introduced in [17], for improving the computation of $S_{Bw}\uparrow\omega$. For reasons of space we present the details of our method for the Backward Strategy only. Its application in the case of the Forward Strategy is similar, and we briefly describe it when presenting our experimental results in Section 5.

The goal 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 in the initial states.

Let us show how our method based on program specialization works on the infinite state reactive system of Example 2.1.

Example 2.2. Let us consider the program *Bw*1 of Example 2.1. The computation of $S_{Bw1}\uparrow\omega$ proceeds as follows:

$S_{Bw1}\uparrow 0 = \{\}$
$S_{Bw1}\uparrow 1 = S_{Bw1}\uparrow 0 \cup \{\mathit{bwReach}(X_1, X_2) \leftarrow 0 > X_1 - X_2\}$ (by clause 3)
$S_{Bw1}\uparrow 2 = S_{Bw1}\uparrow 1 \cup \{\mathit{bwReach}(X_1, X_2) \leftarrow 0 > X_1 - 1\}$ (by clause 2)
$S_{Bw1}\uparrow 3 = S_{Bw1}\uparrow 2 \cup \{\mathit{bwReach}(X_1, X_2) \leftarrow 0 > X_1 + X_2 - 1\}$ (by clause 2)
$S_{Bw1}\uparrow 4 = S_{Bw1}\uparrow 3 \cup \{\mathit{bwReach}(X_1, X_2) \leftarrow 0 > X_1 + 2\,X_2\}$ (by clause 2)
$S_{Bw1}\uparrow 5 = S_{Bw1}\uparrow 4 \cup \{\mathit{bwReach}(X_1, X_2) \leftarrow 0 > X_1 + 3\,X_2 + 2\}$ (by clause 2)
$\ldots$

where, for improving readability, the variables that occur in the constraints but not in the corresponding heads have been projected out. One can easily prove that no new constrained fact in $S_{Bw1} \uparrow k$ will be subsumed by a constrained fact in $S_{Bw1} \uparrow k-1$, for any $k \geq 1$, and thus this construction does not terminate (note that the coefficient of X_2 takes the values $-1, 0, 1, 2, 3, \ldots$).

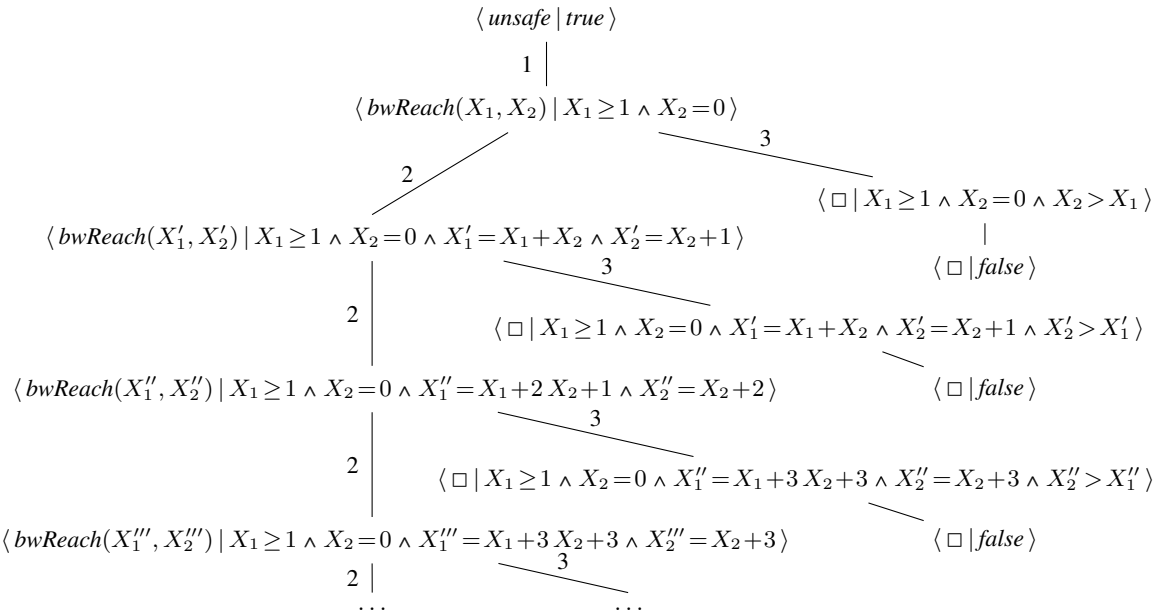

Figure 2. A portion of the tree containing all the evaluations of the query *unsafe*. $\langle\, G \mid c \,\rangle$ denotes the current query G and the corresponding constraint store c. Arcs denote either resolution steps (using the clause whose number is indicated on the arc) or constraint consistency checks. Details on this construction can be found in [22].

In Figure 2 we illustrate (the initial part of) the tree containing all possible top-down evaluations of the query *unsafe*. These top-down evaluations either lead to unsatisfiable constraints or are infinite (note that, in the leftmost branch of the tree, the coefficients of X_2 in the constraint for X_1' are: $1, 2, 3, \ldots$).

These non-termination difficulties can be overcome by specializing the program *Bw*1 with respect to the constraint $X_1 \geq 1 \ \wedge \ X_2 = 0$. Similarly to [17], we apply a specialization technique based on the *unfold/fold* transformation rules for constraint logic programs (see, for instance, [13]). In what follows we illustrate how these transformation rules are applied in order to obtain a specialized version of program *Bw*1. We start off by introducing a new predicate *new*1 defined as follows:

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

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

Now we continue the transformation from the definition of the newly introduced predicate *new*1. 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 the program *Bw*1, and we derive:

6. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge X_1' = X_1 \wedge X_2' = 1 \wedge bwReach(X_1', X_2')$
6′. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge X_2' > X_1$

Clause 6′ is deleted, because its constraint is unsatisfiable. 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 $X_1 \geq 1 \ \wedge \ X_2 = 0 \ \wedge \ X_1' = X_1 \ \wedge \ X_2' = 1$ of clause 6 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 end up introducing, in fact, an infinite sequence of new predicate definitions of the form:

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

with $k = 1, 2, \ldots$ 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. $new1(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 = 0 \wedge X'_1 = X_1 \wedge X'_2 = 1 \wedge new2(X'_1, X'_2)$

We continue the transformation from the definition of the new predicate *new*2. By unfolding clause 8 and then folding again using clause 8 itself 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 specialized program *SpBw*1 is made out of clauses 5, 9, 10, and 11. Now we have that the computation of $S_{SpBw1} \uparrow \omega$ terminates and yields the singleton $\{new2(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 > X_1\}$. Indeed, starting from the empty set, the immediate consequence operator S_{SpBw1} derives the constrained fact $new2(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 > X_1$ (using clause 11) and, from this constrained fact, it does not derive any other constrained fact. Since $unsafe \notin SpBw1$, we have verified that the reactive system of Example 2.1 is safe. □

The form of the specialized program strongly depends on the strategy used to introduce new predicates corresponding to the specialized versions of the predicate *bwReach*. In Example 2.2 we have introduced the two new predicates *new*1 and *new*2 and we have obtained the specialized program by deriving mutually recursive clauses defining those predicates. Note, however, that the definition of *new*2 is *more general than* the definition of *new*1, because the constraint occurring in the body of the clause defining *new*1 implies the constraint occurring in the body of the clause defining *new*2.

By applying an alternative strategy we can introduce *new*2 only and, after folding clause 1 using clause 8, derive a program *SpBw*2 where clauses 5 and 9 are replaced by the single clause:

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

Program *SpBw*2 has fewer clauses than *SpBw*1 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, introducing a set of new predicate definitions that has good performance 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 *SpBw*, and
(iii) ensuring the termination and the efficiency of the computation of $S_{SpBw} \uparrow \omega$.

In general, objectives (i) and (ii) are in contrast with objective (iii), because the introduction of more general definitions obviously improves specialization time and program size, but at the same time reduces the effect of specialization and may not improve the termination behavior for the computation of $S_{SpBw} \uparrow \omega$. In the next section we present a framework for controlling polyvariance and achieving a good balance between the objectives above.

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*

1. UNFOLDING: *Unfold*$(D,$ *UnfD*$)$;
2. 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;
3. 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*$)$

Figure 3. The specialization algorithm.

3. An Algorithm for Controlling Polyvariance During Specialization

Our technique for controlling polyvariance is based on an algorithm for specializing the constraint logic program *Bw* with respect to the constraints characterizing the set of initial states. Our algorithm is *parametric*, in the sense that it depends on three procedures: (1) *Partition*, (2) *Generalize*, and (3) *Fold*. Various definitions of these procedures are provided in Section 4. Depending on the choice of these procedures, our algorithm reduces to various algorithms which have been proposed in the specialization and verification field (see, for instance, [7, 17, 26, 31]), as well as new specialization algorithms which will be discussed in the rest of this paper.

Our parametric specialization algorithm (see Figure 3) constructs a tree, called *DefsTree*, where: (i) each node is labeled by a clause of the form *newp*$(X) \leftarrow d(X) \wedge$ *bwReach*(X), that is, the *definition* of a new predicate introduced by specialization, and (ii) each arc from node D_i to node D_j is labeled by a set of clauses. We will explain below how this set of clauses is constructed. When no confusion arises, we identify a node with its labeling definition. An arc from definition D_i to definition D_j labeled by the set *Cs* of clauses is denoted by $D_i \xrightarrow{Cs} D_j$. For instance, in Example 2.2 the clauses 4, 6, and 8 are related as indicated by the arc $4 \xrightarrow{\{6\}} 8$. A definition D in *DefsTree* is said to be *recurrent* iff D labels both a leaf node and a non-leaf node of *DefsTree*. 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. The algorithm constructs the children of a non-recurrent definition D in the definition tree *DefsTree* constructed so far, according to the following three steps:

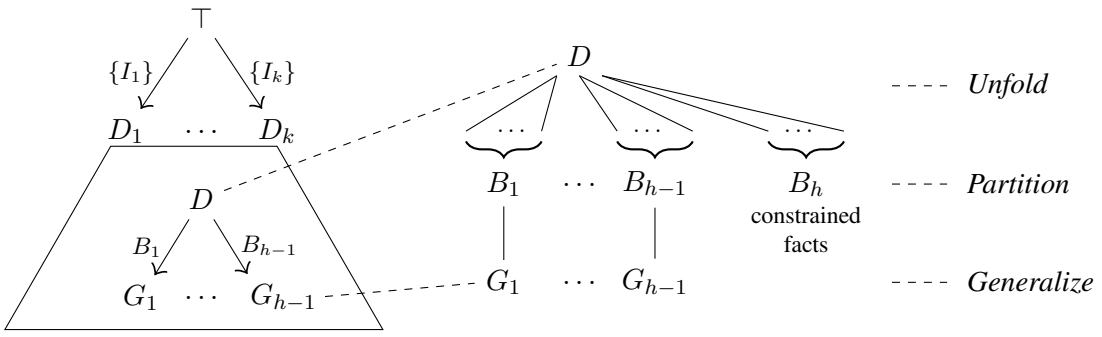

Figure 4. Construction of *DefsTree*, where a generic step of the construction of the children G_1,...,G_{h-1} of a given definition D, by using the procedures *Unfold*, *Partition*, and *Generalize*, is detailed.

1. UNFOLDING: it replaces the atom *bwReach*(X) occurring in the body of the definition D by the bodies of the clauses $T_1, \ldots, T_m, U_1, \ldots, U_n$ that define *bwReach* in *Bw*, thereby deriving, by using the procedure *Unfold*, the following set *UnfD* of $m+n$ clauses:

$$\begin{aligned} \mathit{UnfD} = \{\ & \mathit{newp}(X) \leftarrow c(X) \wedge t_1(X, X') \wedge \mathit{bwReach}(X'),\ \ldots, \\ & \mathit{newp}(X) \leftarrow c(X) \wedge t_m(X, X') \wedge \mathit{bwReach}(X'), \\ & \mathit{newp}(X) \leftarrow c(X) \wedge u_1(X),\ \ldots, \\ & \mathit{newp}(X) \leftarrow c(X) \wedge u_n(X)\ \} \end{aligned}$$

2. CLAUSE REMOVAL: it removes from *UnfD* the clauses whose body contains an unsatisfiable constraint and the clauses that are *subsumed* by a (distinct) constrained fact in *UnfD*.

Note that, the clause removal procedure does not affect the termination of the specialization algorithm, while it can affect both the specialization time and the efficiency of the specialized program. In particular, we could choose either to apply subsumption to any pair of clauses (not requiring one to be a constrained fact) or not to apply it at all. In practice, experiments show that subsumption with respect to constrained facts determines a good balance between specialization time and efficiency of the specialized programs.

3. DEFINITION INTRODUCTION: it constructs the new definitions $G_1, \ldots, G_h$ which are the children of D using some specific procedures for *Partition* and *Generalize*, as indicated in the following Points (i) and (ii).

(i) The procedure *Partition* computes a set $\{B_1, \ldots, B_h\}$ of pairwise disjoint sets of clauses, called *blocks*, such that *UnfD* $= B_1 \cup \ldots \cup B_h$. In [31] it has been argued that applying generalization to blocks of clauses, instead of single clauses, can increase the precision of program specialization.

(ii) The procedure *Generalize* is applied to each block B_i of the partition and produces a new definition G_i. The main objective of this procedure is to enforce the termination of the specialization process by introducing a finite set of new definitions. More specifically, in order to generate the definition G_i, 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. If all clauses in B_i are constrained facts (and thus no folding step is required), then G_i is the special clause 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 $c(X, X') \sqsubseteq d(X')$. Finally, for $i = 1, \ldots, h$, we add to *DefsTree* the arc $D \xrightarrow{B_i} G_i$.

By construction, the constraint occurring in the body of G_i is entailed by every constraint occurring in

the body of a proper clause in B_i, and hence every proper clause in B_i can be folded using G_i. However, we postpone the folding steps until the end of the construction of the whole tree *DefsTree*. In Figure 4 we show how the children nodes $G_1,\ldots,G_{h-1}$ of a given definition D are constructed, according to the specialization algorithm.

The construction of *DefsTree* terminates when *all* leaf clauses of the current *DefsTree* are recurrent. In general, the 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 [17, 26, 31]).

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

In Section 4 we will show how the two different outcomes of the specialization process presented in Example 2.2 can be obtained by using different instances of our parametric specialization algorithm. In that section we will also indicate how by suitable choices of the *Partition* and *Generalize* procedures we can reconstruct known techniques for controlling generalization and polyvariance [7, 17, 26, 31].

The correctness of our parametric specialization algorithm follows from the correctness of the unfolding, folding, and clause removal rules [13]. The correctness proof is based on the fact that the definitions introduced by the *Generalize* procedure are unfolded at least once before they are used for folding.

Theorem 3.1. (Correctness of the Specialization Algorithm)
Let *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 *unsafe* $\in M($*Bw*$)$ iff *unsafe* $\in M($*SpBw*$)$.

4. Partition, Generalize, and Fold Procedures

In this section we provide several definitions of the *Partition*, *Generalize*, and *Fold* procedures that can be used in our parametric specialization algorithm.

First we observe that the set of all conjunctions of equalities on $\mathbb{D}$ is a finite lattice under the partial order defined by the entailment relation $\sqsubseteq$. Then, let us define the *most specific generalization* $\gamma(c_1,\ldots,c_n)$ of the constraints $c_1,\ldots,c_n$ as the conjunction of: (i) the least upper bound of the conjunctions $fd(c_1),\ldots,fd(c_n)$ of equalities on $\mathbb{D}$, and (ii) the *convex hull* [7] 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 $i = 1,\ldots,n$. For $i = 1,\ldots,n$, we have that $c_i \sqsubseteq \gamma(c_1,\ldots,c_n)$. Note that our notion of most specific generalization is an extension of the one commonly used in the specialization of logic programs [26].

Given a set of constraints $Cs = \{c_1,\ldots,c_n\}$, we define two equivalence relations, $\simeq_{fd}$ and $\simeq_{re}$, on *Cs*. The relation $\simeq_{fd}$ on *Cs*, relates constraints that share their *unique* solution on $\mathbb{D}$: in particular, for every $c_i, c_j \in Cs$, $c_i \simeq_{fd} c_j$ iff $fd(c_i) \equiv fd(c_j)$. The relation $\simeq_{re}$ on *Cs* is defined in two steps: in the first step we define a reflexive, symmetric relation $\downarrow_{\mathbb{R}}$ on *Cs*, relating constraints that share *some* solutions on $\mathbb{R}$, then we define the equivalence relation $\simeq_{re}$ on *Cs* as the transitive closure of $\downarrow_{\mathbb{R}}$ on *Cs*. In particular, for every $c_i, c_j \in Cs$, $c_i \downarrow_{\mathbb{R}} c_j$ iff $re(c_i) \wedge re(c_j)$ 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 a set *UnfD* $:= \{C_1, \ldots, C_n\}$ of clauses such that, 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 blocks $B_1, \ldots, B_{h-1}$ are computed by using one of the following five *partition operators* which we now define. In these definitions, for $i = 1, \ldots, n$, we denote by ct'_i the constraint $ct(C_i)|_{X'}$, that is, the projection of the constraint $ct(C_i)$ of the clause C_i with respect to the variables X'.

(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, $ct'_j \simeq_{fd} ct'_k$ on $\{ct'_1, \ldots, ct'_m\}$;

(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 $C_j, \ldots, C_k$ of clauses in *UnfD* such that for any two consecutive clauses in the sequence, the conjunction of the real constraints on the primed variables is satisfiable, that is, $ct'_j \simeq_{re} ct'_k$ on $\{ct'_1, \ldots, ct'_m\}$;

(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, $ct'_j \simeq_{fd} ct'_k$ and $ct'_j \simeq_{re} ct'_k$ on $\{ct'_1, \ldots, ct'_m\}$;

(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, let B be the set $\{E_1, \ldots, E_k\}$ of clauses and G be obtained as follows.

Step 1. Let $b(X')$ be the most specific generalization $\gamma(ct(E_1)|_{X'}, \ldots, ct(E_k)|_{X'})$.
if in *DefsTree* there is a nearest ancestor A_1 of D (possibly D itself) of the form (modulo variable renaming) *newq*$(X') \leftarrow a_1(X') \wedge$ *bwReach*(X') such that $a_1(X') \simeq_{fd} ct(D)$
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 the operators *Widen* and *WidenSum* defined below and other operators defined in [17]).
if in *DefsTree* there is a clause H of the form (modulo variable renaming)
newt$(X') \leftarrow d(X') \wedge$ *bwReach*(X') such that $b_{anc}(X') \sqsubseteq d(X')$
then G is H
else let *newu* be a new predicate symbol

if in *DefsTree* there exists a nearest ancestor A_2 of D (possibly D itself) of the form (modulo variable renaming) $\mathit{newr}(X') \leftarrow a_2(X') \wedge \mathit{bwReach}(X')$ such that $a_2(X') \simeq_{fd} b_{anc}(X')$
then G is $\mathit{newu}(X') \leftarrow (a_2(X') \ominus b_{anc}(X')) \wedge \mathit{bwReach}(X')$
else G is $\mathit{newu}(X') \leftarrow b_{anc}(X') \wedge \mathit{bwReach}(X')$.

In [17] 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 will report in the next section. Indeed, as indicated in [17], these two operators perform better than the other operators considered in [17].

Widen. For any two constraints c and d, where $c = a_1 \wedge \ldots \wedge a_m$ and 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 [7] for a similar widening operator used in static analysis).

WidenSum. We start by defining the *thin well-quasi ordering* $\precsim_S$ (see [10] for this notion). For any atomic constraint a on $\mathbb{R}$ of the form $q_0 + q_1X_1 + \ldots + q_kX_k < 0$ or $q_0 + q_1X_1 + \ldots + q_kX_k \leq 0$, we define $\mathit{sumcoeff}(a)$ to be $\sum_{j=0}^{k} |q_j|$. Given the constraints a_1 of the form $p_1 < 0$ and a_2 of the form $p_2 < 0$, we define $a_1 \precsim_S a_2$ iff $\mathit{sumcoeff}(a_1) \leq \mathit{sumcoeff}(a_2)$. Similarly, given a_1 of the form $p_1 \leq 0$ and a_2 of the form $p_2 \leq 0$, we define $a_1 \precsim_S a_2$ iff $\mathit{sumcoeff}(a_1) \leq \mathit{sumcoeff}(a_2)$.
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 $\{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\} \cup \{b_k \mid b_k \text{ occurs in } \mathit{re}(d) \text{ and there exists } a_i \text{ occurring in } \mathit{re}(c) \text{ such that } b_k \precsim_S a_i\}$, which is the set of atomic constraints that *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 $\precsim_S$.

Our *Partition* and *Generalize* procedures can be suitably chosen so as to obtain specialization or abstract interpretation algorithms already available in the literature. In particular, (i) the technique proposed by Cousot and Halbwachs [7] can be obtained by using the operators *FiniteDomain* and *Widen*, (ii) the specialization algorithm by Peralta and Gallagher [31] can be obtained by using the operators *All* and *Widen*, and (iii) the technique presented in [17] can be obtained by using the partition operator *Singleton* together with the generalization operators *Widen* or *WidenSum*.

FOLD. We define a notion of ordering between definitions. Consider the definitions $C : \mathit{newp}(X) \leftarrow c(X) \wedge \mathit{bwReach}(X)$ and $D : \mathit{newq}(X) \leftarrow d(X) \wedge \mathit{bwReach}(X)$, we say that C is *more general than* D, and we write $D \sqsubseteq C$, iff $d(X) \sqsubseteq c(X)$. A definition C is said to be *maximally general* in a set S of definitions iff, for every definition $D \in S$, if $C \sqsubseteq D$ then $D \sqsubseteq C$. (Note that the relation $\sqsubseteq$ is not antisymmetric.) For the *Fold* procedure we have 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 proper clause E in F such that E occurs in the block B_i labeling 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 proper 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. However, the *Maximally General*

Fold procedure allows us to use a (possibly strict) subset of the definitions introduced by the specialization algorithm, thereby reducing polyvariance and the size of the specialized program. Note 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.

As already mentioned in the previous section, the specialization algorithm which we have applied in Example 2.2 can be obtained by instantiating our parametric specialization algorithm using the following operators: *Singleton* for partitioning, *Widen* for generalization, and *Immediate Fold* for folding. The alternative program mentioned after Example 2.2 can be obtained using the following operators: *Singleton* for partitioning, *Widen* for generalization, and *Maximally General Fold* for folding.

5. Experimental Evaluation

We have implemented the parametric specialization algorithm presented in Section 3 using MAP [29], an experimental system for transforming constraint logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `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. In [29] is provided a web interface for using the MAP system, it is possible to retrieve all the experiments of this paper, and also run them on the same machine used for the benchmarks.

We have performed the backward and forward reachability analyses of several infinite state reactive systems taken from the literature [1, 2, 4, 9, 24, 32]. Among others, we have considered some mutual exclusion and cache coherence protocols, client-server systems, producer-consumer systems, array bound checking, and Reset Petri nets. All the systems considered here satisfy their safety conditions.

For backward reachability we have applied the method presented in Section 2. For forward reachability we have applied a variant of that method: first, (i) we have encoded the forward reachability algorithm by a constraint logic program *Fw* [16] 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}.

In Tables 1 and 2 we report 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.

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 on program *Fw* for forward reachability). For each example the tables have two rows corresponding, respectively, to the *Immediate Fold* procedure (*Im*) and *Maximally General Fold* procedure (*MG*). In the six rightmost columns, we have shown the total verification time (that is, the sum of the specialization time and the time taken for computing the least fixpoint of the immediate consequence operator of the specialized program) 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*. In Tables 1 and 2, the last row, called *no. of successes*, indicates for each column the *precision* (that is, the number of successful verifications) of the corresponding combinations of the partition and generalization operators. Those rows contain two values, one for the *Immediate Fold* procedure (*Im*) and one for the *Maximally General*

Fold procedure (*MG*).

The symbol '∞' means that either the program specialization or the least fixpoint construction does not terminate within 300 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.

If we consider precision, reported in the last two rows of Tables 1 and 2, 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*). The comparison of the *Input* columns of Tables 1 and 2 shows that, on the chosen set of benchmarks, the computation of $S_{Bw} \uparrow \omega$ terminates more often that the computation of $S_{Fw} \uparrow \omega$. This may motivate the better improvements obtained by specialization, when applied to forward reachability rather than to backward reachability. However, in general, there is no reason why specialization should be more effective in the case of forward reachability.

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 (namely, 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* usually determines more polyvariance than *Widen*. If we consider the verification times, they are generally favorable to *WidenSum* with respect to *Widen*.

If we compare the partition operators we have that *All* is strictly less precise than all other operators: it successfully terminates in 138 cases out of 232. Those cases are obtained by varying: (i) the given infinite state system, (ii) the property to be verified (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 example. 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 (namely, forward reachability of Peterson, InsertionSort and SelectionSort) which can only be proved using *Singleton*.

The two operators *Singleton* and *FDC* determine similar degrees of polyvariance and similar verification times, while the operator *All* yields a specialized program with lower degree of polyvariance than *Singleton* and *FDC*. On average, the polyvariance introduced by the generalization operators pays off both in terms of precision and in terms of reduced verification time.

Regarding the two *Fold* procedures, we have an opposite behavior: except for a few cases (namely, backward reachability of Bakery4, Peterson and CSM), the reduction of the degree of polyvariance due to *Maximally General Fold* does not affect precision. Therefore, since it often allows a faster fixpoint construction than *Immediate Fold*, the reduction of polyvariance in the FOLDING phase has to be preferred.

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

	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	70 (3)	110 (14)	100 (14)	90 (5)	30 (15)	20 (15)
Bakery3	2710	*Im*	7240 (5)	3790 (144)	3870 (144)	1100 (6)	200 (77)	150 (77)
		MG	3180 (3)	2650 (60)	2210 (58)	810 (5)	230 (53)	200 (53)
Bakery4	129900	*Im*	∞	112340 (535)	111540 (539)	19340 (6)	102140 (1745)	101300 (1745)
		MG	138210 (3)	38940 (272)	37160 (275)	13150 (5)	78010 (534)	76640 (534)
MutAst	1220	*Im*	4370 (6)	350 (173)	330 (173)	7850 (7)	170 (112)	150 (112)
		MG	1390 (3)	330 (59)	340 (59)	2010 (3)	170 (71)	160 (71)
Peterson N	166520	*Im*	∞	∞	∞	620 (9)	260 (22)	220 (22)
		MG	∞	∞	170510 (3)	690 (6)	320 (22)	300 (22)
Ticket	∞	*Im*	∞	30 (11)	10 (11)	∞	20 (11)	20 (11)
		MG	∞	10 (11)	20 (11)	∞	30 (11)	10 (11)
Berkeley RISC	20	*Im*	80 (5)	70 (6)	30 (6)	70 (5)	50 (8)	40 (8)
		MG	80 (3)	70 (3)	20 (3)	80 (4)	40 (6)	30 (6)
DEC Firefly	50	*Im*	140 (5)	160 (7)	100 (7)	320 (7)	30 (6)	20 (6)
		MG	140 (3)	170 (3)	90 (3)	280 (4)	30 (5)	20 (5)
IEEE Futurebus+	14890	*Im*	16900 (6)	45240 (14)	44340 (14)	16910 (6)	2580 (19)	2410 (19)
		MG	15140 (3)	16860 (3)	14860 (3)	15150 (3)	1780 (6)	1500 (6)
Illinois University	70	*Im*	210 (5)	150 (7)	60 (7)	110 (5)	30 (6)	20 (6)
		MG	190 (3)	140 (4)	60 (4)	100 (3)	30 (5)	20 (5)
MESI	60	*Im*	120 (5)	50 (6)	50 (6)	90 (5)	40 (7)	20 (7)
		MG	110 (3)	60 (4)	30 (4)	90 (4)	30 (5)	20 (5)
MOESI	50	*Im*	220 (6)	190 (7)	130 (7)	250 (6)	90 (7)	50 (7)
		MG	200 (3)	150 (3)	90 (3)	190 (3)	80 (4)	40 (4)
Synapse N+1	10	*Im*	30 (4)	20 (5)	10 (5)	30 (4)	20 (5)	20 (5)
		MG	20 (3)	30 (4)	10 (4)	30 (3)	30 (4)	10 (4)
Xerox PARC Dragon	80	*Im*	230 (5)	180 (7)	80 (7)	470 (7)	60 (8)	30 (8)
		MG	210 (3)	170 (4)	60 (4)	460 (4)	70 (6)	30 (6)
Barber	420	*Im*	290 (5)	5170 (31)	3210 (35)	750 (6)	900 (44)	300 (43)
		MG	270 (3)	3120 (6)	690 (6)	350 (3)	960 (14)	340 (20)
Bounded Buffer	20	*Im*	170 (5)	400 (11)	280 (11)	210 (6)	4490 (75)	3230 (75)
		MG	150 (3)	310 (3)	180 (3)	200 (4)	4000 (18)	2890 (21)
Unbounded Buffer	20	*Im*	100 (6)	200 (12)	150 (12)	70 (6)	210 (12)	130 (12)
		MG	80 (3)	140 (4)	100 (4)	60 (3)	150 (4)	110 (4)
CSM	188110	*Im*	∞	∞	∞	∞	9870 (146)	6920 (154)
		MG	196100 (3)	206880 (3)	189090 (3)	∞	10780 (22)	7850 (30)
Insertion Sort	40	*Im*	90 (7)	60 (23)	60 (23)	130 (8)	90 (28)	80 (28)
		MG	100 (5)	50 (9)	50 (9)	130 (5)	110 (12)	100 (12)
Selection Sort	∞	*Im*	∞	∞	∞	∞	220 (35)	170 (32)
		MG	∞	∞	∞	∞	290 (17)	200 (17)
Office Light Control	20	*Im*	60 (5)	20 (9)	10 (9)	50 (5)	20 (9)	20 (9)
		MG	60 (3)	10 (7)	20 (7)	50 (3)	20 (7)	10 (7)
Reset Petri Nets	∞	*Im*	∞	∞	∞	20 (5)	10 (5)	20 (5)
		MG	∞	∞	∞	20 (3)	20 (3)	10 (3)
GB	1750	*Im*	4780 (6)	3300 (10)	3300 (10)	6520 (6)	2190 (10)	2190 (10)
		MG	1770 (3)	1850 (4)	1850 (4)	1810 (3)	2000 (5)	2010 (5)
Kanban	∞	*Im*	∞	∞	∞	∞	8310 (162)	8170 (162)
		MG	∞	∞	∞	∞	13790 (162)	13690 (162)
McCarthy 91	∞	*Im*	∞	∞	∞	4130 (104)	∞	∞
		MG	∞	∞	∞	3860 (3)	∞	∞
Scheduler	∞	*Im*	4020 (5)	5770 (20)	5700 (20)	17530 (7)	3220 (91)	3120 (91)
		MG	2330 (3)	2610 (6)	2420 (6)	12890 (3)	3360 (15)	3270 (15)
Train	∞	*Im*	1710 (6)	1340 (14)	1330 (14)	3030 (8)	20250 (299)	19850 (299)
		MG	1440 (4)	970 (6)	950 (6)	2450 (5)	14010 (81)	12790 (80)
TTP	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Consistency	∞	*Im*	∞	∞	∞	350 (13)	160 (20)	160 (21)
		MG	∞	∞	∞	380 (7)	200 (14)	150 (14)
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	*Fold*	*All_W*	*FDC_W*	*Single_W*	*All_WS*	*FDC_WS*	*Single_WS*
Bakery2	∞	*Im*	20 (5)	∞	∞	30 (5)	20 (20)	20 (20)
		MG	30 (4)	∞	∞	30 (4)	30 (15)	20 (15)
Bakery3	∞	*Im*	∞	∞	∞	∞	1380 (223)	1190 (240)
		MG	∞	∞	∞	∞	1530 (131)	1360 (134)
Bakery4	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
MutAst	370	*Im*	420 (4)	1790 (190)	1720 (190)	410 (4)	280 (141)	280 (141)
		MG	410 (3)	760 (51)	710 (51)	400 (3)	330 (123)	320 (123)
Peterson N	630	*Im*	∞	∞	1220 (6)	∞	∞	8000 (80)
		MG	∞	∞	710 (3)	∞	∞	8650 (49)
Ticket	50	*Im*	60 (4)	240 (30)	210 (30)	60 (4)	210 (26)	180 (26)
		MG	60 (3)	220 (11)	180 (11)	50 (3)	220 (16)	200 (16)
Berkeley 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)	20 (4)
DEC Firefly	∞	*Im*	110 (3)	130 (3)	∞	110 (3)	100 (3)	60 (9)
		MG	120 (3)	120 (3)	∞	120 (3)	110 (3)	60 (7)
IEEE Futurebus+	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Illinois University	∞	*Im*	150 (3)	150 (3)	∞	140 (3)	150 (3)	70 (8)
		MG	140 (3)	160 (3)	∞	150 (3)	150 (3)	60 (6)
MESI	∞	*Im*	90 (3)	90 (3)	∞	90 (3)	90 (3)	∞
		MG	80 (3)	90 (3)	∞	90 (3)	90 (3)	∞
MOESI	∞	*Im*	130 (3)	130 (3)	∞	130 (3)	130 (3)	∞
		MG	120 (3)	140 (3)	∞	130 (3)	130 (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 PARC Dragon	∞	*Im*	180 (3)	190 (3)	∞	190 (3)	210 (3)	80 (8)
		MG	190 (3)	210 (3)	∞	180 (3)	210 (3)	90 (6)
Barber	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Bounded Buffer	∞	*Im*	∞	50 (4)	20 (4)	∞	50 (4)	20 (4)
		MG	∞	50 (4)	20 (4)	∞	50 (4)	20 (4)
Unbounded Buffer	∞	*Im*	∞	210 (8)	70 (8)	∞	190 (8)	70 (8)
		MG	∞	220 (6)	80 (6)	∞	220 (6)	80 (6)
CSM	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Insertion Sort	∞	*Im*	∞	∞	10 (14)	∞	∞	20 (14)
		MG	∞	∞	20 (14)	∞	∞	20 (14)
Selection Sort	∞	*Im*	∞	∞	180 (37)	∞	∞	310 (47)
		MG	∞	∞	230 (24)	∞	∞	430 (34)
Office Light Control	∞	*Im*	∞	30 (18)	20 (18)	∞	30 (18)	20 (18)
		MG	∞	20 (16)	20 (16)	∞	30 (16)	20 (16)
Reset Petri Nets	∞	*Im*	∞	∞	∞	0 (6)	10 (6)	0 (6)
		MG	∞	∞	∞	0 (5)	10 (5)	10 (5)
GB	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Kanban	44860	*Im*	46840 (4)	46860 (4)	56100 (13)	∞	∞	∞
		MG	43630 (3)	43670 (3)	43660 (3)	∞	∞	∞
McCarthy 91	∞	*Im*	∞	∞	∞	∞	∞	∞
		MG	∞	∞	∞	∞	∞	∞
Scheduler	840	*Im*	910 (3)	910 (4)	1750 (32)	930 (3)	920 (4)	127370 (530)
		MG	920 (3)	920 (4)	1120 (4)	930 (3)	940 (4)	70510 (80)
Train	∞	*Im*	∞	∞	∞	∞	∞	410 (51)
		MG	∞	∞	∞	∞	∞	720 (38)
TTP	∞	*Im*	∞	∞	∞	650 (4)	1140 (15)	∞
		MG	∞	∞	∞	670 (4)	1190 (11)	∞
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.

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, [7, 17, 26, 31]).

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 provided, 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 [8, 17, 26, 30, 31]. As already mentioned, the techniques presented in [17, 31] can be considered as instances of our framework, while [26, 30] do not consider constraints, which are of primary concern here. Our framework generalizes and improves the framework of [17], by introducing partitioning and folding operators which, as shown in Section 5, increase the precision and the performance of the verification process. Note that in [17] we considered a subset of the experiments addressed in the present paper. The results obtained in [17] can be obtained in the framework proposed here by using the *Singleton* strategy for partitioning and the *Immediate Fold* strategy for folding. Here we address only a subset of the generalization operators illustrated in that paper: they are the *Widen* and *WidenSum* operators, that perform well on our benchmarks. Also the specialization technique presented in [16] can be obtained by using the *Singleton* and the *Immediate Fold* strategies presented here. That technique makes use of program specialization as a preprocessing step for further analyses performed by applying verification tools for counter systems such as ALV [32].

The *off-line specialization* approach followed by [8] is based on a preliminary *binding time analysis* to decide when to unfold a predicate 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 further 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. Preliminary results in this direction can be found in [11], where our specialization technique is applied and adapted to the analysis of C programs.

Acknowledgements

This work has been partially supported by the MIUR PRIN project n.20089M932N "Innovative and multi-disciplinary approaches for reasoning with constraints and preferences" and by the joint CNR (Italy)/CNRS (France) project: "Verification of infinite state and real time systems". Many thanks to Laurent Fribourg and John Gallagher for stimulating conversations on the topics of this paper, and to the

anonymous referees for constructive criticism.

References

[1] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Proc. 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 *Proc. 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 *Proc. 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. *Int. J. 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] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV 2000*, Lecture Notes in Computer Science 1855, pages 154–169. Springer, 2000.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.

[8] 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.

[9] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[10] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Software Model Checking by Program Specialization. In *Proc. of the 9th Italian Convention on Computational Logic (CILC'12)*, CEUR-WS Vol.857. 2012.

[12] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[13] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[14] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Proc. of LOPSTR '00*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.

[15] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proc. of VCL'01*, Tech. Rep. DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.

[16] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving Reachability Analysis of Infinite State Systems by Specialization. In *Proc. of RP 2011*, Lecture Notes in Computer Science 6945, pages 165–179. Springer, 2011.

[17] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization Strategies for the Verification of Infinite State Systems. *Theory and Practice of Logic Programming*. Available on CJO doi:10.1017/S1471068411000627. Cambridge, 2012.

[18] 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.

[19] L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, editor, *Proc. 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.

[20] J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[21] T. J. Hickey and D. A. Smith. Towards the partial evaluation of CLP languages. In *Proc. 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.

[22] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.

[23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[24] LASH. homepage: `http://www.montefiore.ulg.ac.be/~boigelot/research/lash` .

[25] 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.

[26] 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.

[27] 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.

[28] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[29] MAP. The MAP transformation system web interface: `http://map.uniroma2.it/mapweb` .

[30] C. Ochoa, G. Puebla, and M. V. Hermenegildo. Removing superfluous versions in polyvariant specialization of Prolog programs. In *Proc. of LOPSTR '05*, Lecture Notes in Computer Science 3961, pages 80–97. Springer, 2006.

[31] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *Proc. of LOPSTR '02*, Lecture Notes in Computer Science 2664, pages 90–108. Springer, 2003.

[32] 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.