Improving Reachability Analysis of Infinite State Systems by Specialization

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

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

Abstract. We consider infinite state reactive systems specified by using linear constraints over the integers, and we address the problem of verifying safety properties of these systems by applying reachability analysis techniques. We propose a method based on program specialization, which improves the effectiveness of the backward and forward reachability analyses. For backward reachability our method consists in: (i) specializing the reactive system with respect to the initial states, and then (ii) applying to the specialized system a reachability analysis that works backwards from the unsafe states. For forward reachability our method works as for backward reachability, except that the role of the initial states and the unsafe states are interchanged. We have implemented our method using the MAP transformation system and the ALV verification system. Through various experiments performed on several infinite state systems, we have shown that our specialization-based verification technique considerably increases the number of successful verifications without significantly degrading the time performance.

1 Introduction

One of the present challenges in the field of automatic verification of reactive systems is the extension of the 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 [10] for a survey of relevant results).

In order to overcome this limitation, several authors have advocated the use of *constraints* over the integers (or the reals) to represent infinite sets of states [4,8,9,15,17]. By manipulating constraint-based representations of sets of states, one can verify a safety property φ of an infinite state system by one of the following two strategies:
(i) 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 (that is, a state where $\neg\varphi$ holds), and then one checks whether or not BR has an empty intersection with the set I of the initial states;

CONCURRENCY, SPECIFICATION AND PROGRAMMING
M. Szczuka et al. (eds.): Proceedings of the international workshop CS&P 2011
September 28-30, Pułtusk, Poland, pp. 135-148

(ii) 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 [2,3,14,20,25]. Some of them also use techniques borrowed from the field of *abstract interpretation* [6], whereby in order to check whether or not a safety property φ holds for all states which are reachable from the initial states, an *upper approximation* $\overline{BR}$ (or $\overline{FR}$) of the set *BR* (or *FR*) is computed. These techniques improve the termination of the verification tools at the expense of a possible loss in precision. Indeed, whenever $\overline{BR} \cap I \neq \emptyset$ (or $\overline{FR} \cap U \neq \emptyset$), one cannot conclude that, for some state, φ does not hold.

One weakness of the Backward Strategy is that, when computing *BR*, it does not take into account the properties holding on the initial states. This may lead, even if the formula φ does hold, to a failure of the verification process, because either the computation of *BR* does not terminate or one gets an overly approximated $\overline{BR}$ with a non-empty intersection with the set I. 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* or $\overline{FR}$.

In this paper we present a method, based on *program specialization* [19], for overcoming these weaknesses. 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. Our specialization method is applied before computing *BR* (or *FR*). Its objective is to transform the constraint-based specification of a reactive system into a new specification that, when used for computing *BR* (or *FR*), takes into consideration also the properties holding on the initial states (or the unsafe states, respectively).

Our method consists of the following three steps: (1) the translation of a reactive system specification into a *constraint logic program* (CLP) [18] that implements backward (or forward) reachability; (2) the specialization of the CLP program with respect to the initial states (or the unsafe states, respectively), and (3) the reverse translation of the specialized CLP program into a specialized reactive system. We prove that our specialization method is correct, that is, it transforms a given specification into one which satisfies the same safety properties.

We have implemented our specialization method on the MAP transformation system for CLP programs [22] and we have performed experiments on several infinite state systems by using the *Action Language Verifier* (ALV) [25]. These experiments show that specialization determines a relevant increase of the number of successful verifications, in the case of both backward and forward reachability analysis, without a significant degradation of the time performance.

2 Specifying Reactive Systems

In order to specify reactive systems and their safety properties, we use a simplified version of the languages considered in [2,3,20,25]. Our language allows us to specify systems and properties by using constraints over the set $\mathbb{Z}$ of the integers.

A *system* is a triple $\langle Var, Init, Trans\rangle$, where: (i) *Var* is a *variable declaration*, (ii) *Init* is a formula denoting the set of *initial states*, and (iii) *Trans* is a formula denoting a *transition relation* between states.

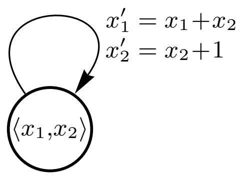
Now we formally define these notions. A variable declaration *Var* is a sequence of declarations of (distinct) variables each of which may be either: (i) an *enumerated* variable, or (ii) an *integer* variable. (i) An enumerated variable x is declared by the statement: **enumerated** x D, meaning that x ranges over a finite set D of constants. The set D is said to be the *type* of x and it is also said to be the type of every constant in D. (ii) An integer variable x is declared by the statement: **integer** x, meaning that x is a variable ranging over the set $\mathbb{Z}$ of the integers. By $\mathcal{X}$ we denote the set of variables declared in *Var*, and by $\mathcal{X}'$ we denote the set $\{x' \mid x \in \mathcal{X}\}$ of primed variables.

Constraints are defined as follows. If e_1 and e_2 are enumerated variables or constants of the same type, then $e_1 = e_2$ and $e_1 \neq e_2$ are *atomic constraints*. If p_1 and p_2 are linear polynomials with integer coefficients, then $p_1 = p_2$, $p_1 \geq p_2$, and $p_1 > p_2$ are *atomic constraints*. A *constraint* is either *true*, or *false*, or an atomic constraint, or a *conjunction* of constraints. *Init* is a disjunction of constraints on the variables in $\mathcal{X}$. *Trans* is a disjunction of constraints on the variables in $\mathcal{X} \cup \mathcal{X}'$.

A *specification* is a pair $\langle Sys, Safe\rangle$, where *Sys* is a system and *Safe* is a formula of the form $\neg\,\mathsf{EF}\,Unsafe$, specifying a *safety property* of the system, and *Unsafe* is a disjunction of constraints on the variables in $\mathcal{X}$.

Example 1. Here we show a reactive system (1.1) and its specification (1.2) in our language.

(1.1)



(1.2)

Var: **integer** x_1; **integer** x_2;
Init: $x_1 \geq 1 \ \wedge\ x_2 = 0$;
Trans: $x_1' = x_1+x_2 \ \wedge\ x_2' = x_2+1$;
Safe: $\neg\,\mathsf{EF}(x_2 > x_1)$

□

Now we define the semantics of a specification. Let D_i be a finite set of constants, for $i = 1, \ldots, k$. Let $X = \langle x_1, \ldots, x_k, x_{k+1}, \ldots, x_n\rangle$ be a listing of the variables in $\mathcal{X}$, where: (i) for $i = 1, \ldots, k$, x_i is an enumerated variable of type D_i, and (ii) for $i = k+1, \ldots, n$, x_i is an integer variable. Let X' be a listing $\langle x_1', \ldots, x_k', x_{k+1}', \ldots, x_n'\rangle$ of the variables in $\mathcal{X}'$. A *state* is an n-tuple $\langle r_1, \ldots, r_k, z_{k+1}, \ldots, z_n\rangle$ of constants in $D_1 \times \ldots \times D_k \times \mathbb{Z}^{n-k}$.

A state s of the form $\langle r_1, \ldots, r_k, z_{k+1}, \ldots, z_n\rangle$ *satisfies* a disjunction d of constraints on $\mathcal{X}$, denoted $s \models d$, if the formula $d[s/X]$ holds, where $[s/X]$ denotes the substitution $[r_1/x_1, \ldots, r_k/x_k, z_{k+1}/x_{k+1}, \ldots, z_n/x_n]$. A state satisfying *Init* (resp., *Unsafe*) will be called an *initial* (resp., *unsafe*) state.

A pair of states $\langle s, s'\rangle$ *satisfies* a constraint c on the variables in $\mathcal{X} \cup \mathcal{X}'$, denoted $\langle s, s'\rangle \models c$, if the constraint $c[s/X, s'/X']$ holds. A *computation sequence* is a sequence of states $s_0, \ldots, s_m$, with $m \geq 0$, such that, for $i = 0, \ldots, m-1$, $\langle s_i, s_{i+1}\rangle \models c$, for some constraint c in $\{c_j \mid j \in J\}$, where $Trans = \bigvee_{j\in J} c_j$.

State s_m is *reachable* from state s_0 if there exists a computation sequence $s_0, \ldots, s_m$. The system *Sys satisfies* the safety property, called *Safe*, of the form $\neg\, \mathsf{EF}\, \mathit{Unsafe}$, if there is no state s which is reachable from an initial state and $s \models \mathit{Unsafe}$.

A specification $\langle \mathit{Sys}_1, \mathit{Safe}_1 \rangle$ is *equivalent* to a specification $\langle \mathit{Sys}_2, \mathit{Safe}_2 \rangle$ if Sys_1 satisfies Safe_1 if and only if Sys_2 satisfies Safe_2.

3 Constraint-Based Specialization of Reactive Systems

Now we present a method for transforming a specification $\langle \mathit{Sys}, \mathit{Safe} \rangle$ into an equivalent specification whose safety property is easier to verify. This method has two variants, called *Bw-Specialization* and *Fw-Specialization*. Bw-Specialization specializes the given system with respect to the disjunction *Init* of constraints that characterize the initial states. Thus, backward reachability analysis of the specialized system may be more effective because it takes into account the information about the initial states. A symmetric situation occurs in the case of Fw-Specialization where the given system is specialized with respect to the disjunction *Unsafe* of constraints that characterize the unsafe states.

Here we present the Bw-Specialization method only. (The Fw-Specialization method is similar and it is described in Appendix.) Bw-Specialization transforms the specification $\langle \mathit{Sys}, \mathit{Safe} \rangle$ into an equivalent specification $\langle \mathit{SpSys}, \mathit{SpSafe} \rangle$ according to the following three steps.

Step (1). *Translation*: The specification $\langle \mathit{Sys}, \mathit{Safe} \rangle$ is translated into a CLP program, called *Bw*, that implements the backward reachability algorithm.

Step (2). *Specialization*: The CLP program *Bw* is specialized into a program *SpBw* by taking into account the disjunction *Init* of constraints.

Step (3). *Reverse Translation*: The specialized CLP program *SpBw* is translated back into a new, specialized specification $\langle \mathit{SpSys}, \mathit{SpSafe} \rangle$, which is equivalent to $\langle \mathit{Sys}, \mathit{Safe} \rangle$.

The specialized specification $\langle \mathit{SpSys}, \mathit{SpSafe} \rangle$ contains new constraints that are derived by propagating through the transition relation of the system *Sys* the constraints *Init* holding in the initial states. Thus, the backward reachability analysis that uses the transition relation of the specialized system *SpSys*, takes into account the information about the initial states and, for this reason, it is often more effective (see Section 4 for an experimental validation of this fact).

Let us now describe Steps (1), (2), and (3) in more detail.

Step (1). Translation. Let us consider the system $\mathit{Sys} = \langle \mathit{Var}, \mathit{Init}, \mathit{Trans} \rangle$ and the property *Safe*. Suppose that:

(1) X and X' are listings of the variables in the sets $\mathcal{X}$ and $\mathcal{X}'$, respectively,

(2) *Init* is a disjunction $\mathit{init}_1(X) \vee \ldots \vee \mathit{init}_k(X)$ of constraints,

(3) *Trans* is a disjunction $t_1(X, X') \vee \ldots \vee t_m(X, X')$ of constraints,

(4) *Safe* is the formula $\neg\, \mathsf{EF}\, \mathit{Unsafe}$, where *Unsafe* is a disjunction $u_1(X) \vee \ldots \vee u_n(X)$ of constraints.

Then, program *Bw* consists of the following clauses:

I_1: $\mathit{unsafe} \leftarrow \mathit{init}_1(X) \wedge \mathit{bwReach}(X)$

$\cdots$

I_k: *unsafe* $\leftarrow init_k(X) \wedge bwReach(X)$
T_1: $bwReach(X) \leftarrow t_1(X, X') \wedge bwReach(X')$
...
T_m: $bwReach(X) \leftarrow t_m(X, X') \wedge bwReach(X')$
U_1: $bwReach(X) \leftarrow u_1(X)$
...
U_n: $bwReach(X) \leftarrow u_n(X)$

The meaning of the predicates defined in the program *Bw* is as follows:
(i) *bwReach*(X) holds iff an unsafe state can be reached from the state X in zero or more applications of the transition relation, and (ii) *unsafe* holds iff there exists an initial state X such that *bwReach*(X) holds.

Example 2. For the system of Example 1 we get the following CLP program:
I_1: *unsafe* $\leftarrow x_1 \geq 1 \wedge x_2 = 0 \wedge bwReach(x_1, x_2)$
T_1: $bwReach(x_1, x_2) \leftarrow x_1' = x_1 + x_2 \wedge x_2' = x_2 + 1 \wedge bwReach(x_1', x_2')$
U_1: $bwReach(x_1, x_2) \leftarrow x_2 > x_1$ □

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

The translation of the specification $\langle Sys, Safe \rangle$ performed during Step (1) is correct in the sense stated by Theorem 1. The proof of this theorem is based on the fact that the definition of the predicate *bwReach* in the program *Bw* is a recursive definition of the reachability relation defined in Section 2.

Theorem 1 (Correctness of Translation). *The system Sys satisfies the formula Safe iff unsafe* $\notin M(Bw)$.

Step (2). **Specialization**. Program *Bw* is transformed into a specialized program *SpBw* such that *unsafe* $\in M(Bw)$ iff *unsafe* $\in M(SpBw)$ by applying the specialization algorithm shown in Figure 1.

This algorithm modifies the initial program *Bw* by propagating the information about the initial states *Init* and it does so by using the *definition introduction*, *unfolding*, *clause removal*, and *folding* rules for transforming constraint logic programs (see, for instance, [11]). In particular, our specialization algorithm: (i) introduces new predicates defined by clauses of the form $newp(X) \leftarrow c(X) \wedge bwReach(X)$, corresponding to specialized versions of the *bwReach* predicate, and (ii) derives mutually recursive definitions of these new predicates by applying the unfolding, clause removal, and folding rules.

An important feature of our specialization algorithm is that the applicability conditions of the transformation rules used by the algorithm are expressed in terms of the unsatisfiability (or entailment) of constraints on the domain $\mathbb{R}$ of the real numbers, instead of the domain $\mathbb{Z}$ of the integer numbers, thereby allowing us to use more efficient constraint solvers (according to the present state-of-the-art solvers). Note that, despite this domain change from $\mathbb{Z}$ to $\mathbb{R}$, the specialized reachability program *SpBw* is *equivalent* to the initial program *Bw* w.r.t. the least $\mathbb{Z}$-model semantics (see Theorem 4 below). This result is based on the correctness

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

INITIALIZATION:
$SpBw := \{J_1, \ldots, J_k\}$, where J_1: $unsafe \leftarrow init_1(X) \wedge newu_1(X)$
$\cdots$
J_k: $unsafe \leftarrow init_k(X) \wedge newu_k(X)$;
$InDefs := \{I'_1, \ldots, I'_k\}$, where I'_1: $newu_1(X) \leftarrow init_1(X) \wedge bwReach(X)$
$\cdots$
I'_k: $newu_k(X) \leftarrow init_k(X) \wedge bwReach(X)$;
$Defs := InDefs$;
while there exists a clause C: $newp(X) \leftarrow c(X) \wedge bwReach(X)$ in *InDefs* *do*
UNFOLDING: $SpC := \{ newp(X) \leftarrow c(X) \wedge t_1(X, X') \wedge bwReach(X'),$
$\cdots$
$newp(X) \leftarrow c(X) \wedge t_m(X, X') \wedge bwReach(X'),$
$newp(X) \leftarrow c(X) \wedge u_1(X),$
$\cdots$
$newp(X) \leftarrow c(X) \wedge u_n(X) \};$
CLAUSE REMOVAL:
while in *SpC* there exist two distinct clauses E and F such that E $\mathbb{R}$-subsumes F or there exists a clause F whose body has a constraint which is not $\mathbb{R}$-satisfiable *do* $SpC := SpC - \{F\}$ *end-while*;
DEFINITION-INTRODUCTION & FOLDING:
while in *SpC* there is a clause E of the form: $newp(X) \leftarrow e(X, X') \wedge bwReach(X')$ *do*
if in *Defs* there is a clause D of the form: $newq(X) \leftarrow d(X) \wedge bwReach(X)$ such that $e(X, X') \sqsubseteq_{\mathbb{R}} d(X')$, where $d(X')$ is $d(X)$ with X replaced by X'
then $SpC := (SpC - \{E\}) \cup \{newp(X) \leftarrow e(X, X') \wedge newq(X')\}$;
else let $Gen(E, Defs)$ be the clause $newr(X) \leftarrow g(X) \wedge bwReach(X)$ where:
(i) *newr* is a predicate symbol not in *Defs* and (ii) $e(X, X') \sqsubseteq_{\mathbb{R}} g(X')$;
$Defs := Defs \cup \{Gen(E, Defs)\}$; $InDefs := InDefs \cup \{Gen(E, Defs)\}$;
$SpC := (SpC - \{E\}) \cup \{newp(X) \leftarrow e(X, X') \wedge newr(X')\}$;
end-while;
$SpBw := SpBw \cup SpC$;
end-while

Fig. 1. The specialization algorithm.

of the transformation rules [11] and on the fact that the unsatisfiability (or entailment) of constraints on $\mathbb{R}$ implies the unsatisfiability (or entailment) of those constraints on $\mathbb{Z}$. For instance, let us consider the rule that removes a clause of the form $H \leftarrow c \wedge B$ if the constraint c is unsatisfiable on the integers. Our specialization algorithm removes the clause if c is unsatisfiable on the reals. Clearly, we may miss the opportunity of removing a clause whose constraint is satisfiable on the reals and unsatisfiable on the integers, thereby deriving a specialized program with redundant satisfiability tests. More in general, the use of constraint solvers on the reals may reduce the specialization time, but may leave in the specialized programs residual satisfiability tests on the integers that should be performed at verification time on the specialized system.

Let us define the notions of $\mathbb{R}$-satisfiability, $\mathbb{R}$-entailment, and $\mathbb{R}$-subsumption that we have used in the specialization algorithm. Let X and X' be n-tuples of variables as indicated in Section 2. The constraint $c(X)$ is $\mathbb{R}$-*satisfiable*, if there exists an n-tuple A in $D_1 \times \ldots \times D_k \times \mathbb{R}^{n-k}$ such that $c(A)$ holds. A constraint $c(X,X')$ $\mathbb{R}$-*entails* a constraint $d(X,X')$, denoted $c(X,X') \sqsubseteq_{\mathbb{R}} d(X,X')$, if for all A, A' in $D_1 \times \ldots \times D_k \times \mathbb{R}^{n-k}$, if $c(A,A')$ holds then $d(A,A')$ holds. (Note that the variables X or X' may be absent from $c(X,X')$ or $d(X,X')$.) Given two clauses of the forms C: $H \leftarrow c(X)$ and D: $H \leftarrow d(X) \wedge e(X,X') \wedge B$, where the constraint $e(X,X')$ and the atom B may be absent, we say that C $\mathbb{R}$-*subsumes* D, if $d(X) \wedge e(X,X') \sqsubseteq_{\mathbb{R}} c(X)$.

As usual when performing program specialization, our algorithm also makes use of a *generalization operator Gen* for introducing definitions of new predicates by generalizing constraints. Given a clause E: $newp(X) \leftarrow e(X,X') \wedge bwReach(X')$ and the set *Defs* of clauses that define the new predicates introduced so far by the specialization algorithm, $Gen(E, Defs)$ returns a clause G of the form $newr(X) \leftarrow g(X) \wedge bwReach(X)$ such that: (i) *newr* is a fresh, new predicate symbol, and (ii) $e(X,X') \sqsubseteq_{\mathbb{R}} g(X')$ (where $g(X')$ is the constraint $g(X)$ with X replaced by X'). Then, clause E is folded by using clause G, thereby deriving $newp(X) \leftarrow e(X,X') \wedge newr(X')$. This transformation step preserves equivalence with respect to the least $\mathbb{Z}$-model semantics. Indeed, $newr(X')$ is equivalent to $g(X') \wedge bwReach(X')$ by definition and, as already mentioned, $e(X,X') \sqsubseteq_{\mathbb{R}} g(X')$ implies that $e(X,X')$ entails $g(X')$ in $\mathbb{Z}$.

The generalization operator we use in our experiments reported in Section 4, is defined in terms of relations and operators on constraints such as *widening* and *well-quasi orders* based on the coefficients of the polynomials occurring in the constraints. For lack of space we will not describe in detail the generalization operator we apply, and we refer to [13,23] for various operators which can be used for specializing constraint logic programs. It will be enough to say that the termination of the specialization algorithm is ensured by the fact that, similarly to the widening operator presented in [6], our generalization operator guarantees that during specialization only a finite number of new predicates is introduced.

Thus, we have the following result.

Theorem 2 (Termination and Correctness of Specialization). (i) *The specialization algorithm terminates.* (ii) *Let program SpBw be the output of the specialization algorithm. Then unsafe* $\in M(Bw)$ *iff unsafe* $\in M(SpBw)$.

Example 3. The following program is obtained as output of the specialization algorithm when it takes as input the CLP program of Example 2:

J_1: $unsafe \leftarrow x_1 \geq 1 \wedge x_2 = 0 \wedge new1(x_1, x_2)$
S_1: $new1(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 = 0 \wedge x_1' = x_1 \wedge x_2' = 1 \wedge new2(x_1', x_2')$
S_2: $new2(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 = 1 \wedge x_1' = x_1 + 1 \wedge x_2' = 2 \wedge new3(x_1', x_2')$
S_3: $new3(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 \geq 1 \wedge x_1' = x_1 + x_2 \wedge x_2' = x_2 + 1 \wedge new3(x_1', x_2')$
V_1: $new3(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 > x_1$ □

Step (3). **Reverse Translation**. The output of the specialization algorithm is a specialized program *SpBw* of the form:

J_1: $unsafe \leftarrow init_1(X) \wedge newu_1(X)$
...
J_k: $unsafe \leftarrow init_k(X) \wedge newu_k(X)$

S_1: $newp_1(X) \leftarrow s_1(X,X') \wedge newt_1(X')$
...
S_m: $newp_m(X) \leftarrow s_m(X,X') \wedge newt_m(X')$
V_1: $newq_1(X) \leftarrow v_1(X)$
...
V_n: $newq_n(X) \leftarrow v_n(X)$

where: (i) $s_1(X,X'),\ldots,s_m(X,X'),v_1(X),\ldots,v_m(X)$ are constraints, and (ii) the (possibly non-distinct) predicate symbols $newu_i$'s, $newp_i$'s, $newt_i$'s, and $newq_i$'s are the new predicate symbols introduced by the specialization algorithm. Let *NewPred* be the set of all of those new predicate symbols.

We derive a new specification $\langle SpSys, SpSafe\rangle$, where *SpSys* is a system of the form $\langle SpVar, SpInit, SpTrans\rangle$, as follows.

(1) Let x_p be a new enumerated variable ranging over the set *NewPred* of predicate symbols introduced by the specialization algorithm.
Let the variable X occurring in the program *SpBw* denote the n-tuple of variables $\langle x_1,\ldots,x_k,x_{k+1},\ldots,x_n\rangle$, where: (i) for $i=1,\ldots,k$, x_i is an enumerated variable ranging over the finite set D_i, and (ii) for $i=k+1,\ldots,n$, x_i is an integer variable.
We define *SpVar* to be the following sequence of declarations of variables:
enumerated x_p *NewPred*;
enumerated x_1 D_1; ...; **enumerated** x_k D_k;
integer x_{k+1}; ...; **integer** x_n.

(2) From clauses $J_1,\ldots,J_k$ we get the disjunction *SpInit* of k constraints, each of which is of the form: $init_i(X) \wedge x_p = newu_i$.

(3) From clauses $S_1,\ldots,S_m$ we get the disjunction *SpTrans* of m constraints, each of which is of the form: $s_i(X,X') \wedge x_p = newp_i \wedge x'_p = newt_i$.

(4) From clauses $V_1,\ldots,V_n$ we get the disjunction *SpUnsafe* of n constraints, each of which is of the form: $v_i(X) \wedge x_p = newq_i$.
SpSafe is the formula $\neg\,\mathsf{EF}\,SpUnsafe$.

The reverse translation of the program *SpBw* into the specification $\langle SpSys, SpSafe\rangle$ is correct in the sense stated by the following theorem.

Theorem 3 (Correctness of Reverse Translation). *The following equivalence holds: unsafe* $\notin M(SpBw)$ *iff SpSys satisfies SpSafe.*

Example 4. The following specialized specification is the result of the reverse translation of the specialized CLP program of Example 3:

SpVar: **enumerated** x_p $\{new1, new2, new3\}$; **integer** x_1; **integer** x_2;
SpInit: $x_1 \geq 1 \wedge x_2 = 0 \wedge x_p = new1$;
SpTrans: $(x_1 \geq 1 \wedge x_2 = 0 \wedge x_p = new1 \wedge x'_1 = x_1 \wedge x'_2 = 1 \wedge x'_p = new2)\ \vee$
$(x_1 \geq 1 \wedge x_2 = 1 \wedge x_p = new2 \wedge x'_1 = x_1{+}1 \wedge x'_2 = 2 \wedge x'_p = new3)\ \vee$
$(x_1 \geq 1 \wedge x_2 \geq 1 \wedge x_p = new3 \wedge x'_1 = x_1{+}x_2 \wedge x'_2 = x_2{+}1 \wedge x'_p = new3)$
SpSafe: $\neg\,\mathsf{EF}(x_1 \geq 1 \wedge x_2 > x_1 \wedge x_p = new3)$

Note that the backward reachability algorithm implemented in the ALV tool [25] is *not* able to verify (within 600 seconds) the safety property of the *initial specification* (see Example 1). Basically, this is due to the fact that working backward from the unsafe states where $x_2 > x_1$ holds, ALV is not able to infer that, for all reachable states, $x_2 \geq 0$ holds. The Bw-Specialization method is able to derive, from the constraint characterizing the initial states, a new transition relation *SpTrans* whose constraints imply $x_2 \geq 0$. By exploiting this constraint, ALV successfully verifies the safety property of the *specialized specification*. □

The correctness of our Bw-Specialization method is stated by the following theorem, which is a straightforward consequence of Theorems 1, 2, and 3.

Theorem 4 (Correctness of Bw-Specialization). *Let* $\langle SpSys, SpSafe \rangle$ *be the specification derived by applying the Bw-Specialization method to the specification* $\langle Sys, Safe \rangle$*. Then,* $\langle Sys, Safe \rangle$ *is equivalent to* $\langle SpSys, SpSafe \rangle$*.*

4 Experimental Evaluation

In this section we present the results of the verification experiments we have performed on various infinite state systems taken from the literature [3,8,9,25].

We have run our experiments by using the ALV tool, which is based on a BDD-based symbolic manipulation for enumerated types and on a solver for linear constraints on integers [25]. ALV performs backward and forward reachability analysis by an approximate computation of the least fixpoint of the transition relation of the system. We have run ALV using the options: 'default' and 'A' (both for backward analysis), and the option 'F' (for forward analysis). The Bw-Specialization and the Fw-Specialization methods were implemented on MAP [22], a tool for transforming CLP programs which uses the SICStus Prolog `clpr` library to operate on constraints on the reals. All experiments were performed on an Intel Core 2 Duo E7300 2.66 GHz under Linux.

The results of our experiments are reported in Table 1, where we have indicated, for each system and for each ALV option used, the following times expressed in seconds: (i) the time taken by ALV for verifying the given system (columns *Sys*), and (ii) the total time taken by MAP for specializing the system and by ALV for verifying the specialized system (columns *SpSys*).

The experiments show that our specialization method always increases the *precision* of ALV, that is, for every ALV option used, the number of properties verified increases when considering the specialized systems (columns *SpSys*) instead of the given, non-specialized systems (columns *Sys*). There are also some examples (Consistency, Selection Sort, and Reset Petri Net) where ALV is not able to verify the property on the given reactive system (regardless of the option used), but it verifies the property on the corresponding specialized system.

Now, let us compare the verification times. The time in column *Sys* and the time in column *SpSys* are of the same order of magnitude in almost all cases. In two examples (Peterson and CSM, with the 'default' option) our method substantially reduces the total verification time. Finally, in the Bounded Buffer example (with options 'default' and 'A') our specialization method significantly increases the verification time. Thus, overall, the increase of precision due to the specialization method we have proposed, does not determine a significant degradation of the time performance.

	default		A		F	
EXAMPLES	*Sys*	*SpSys*	*Sys*	*SpSys*	*Sys*	*SpSys*
1. Bakery2	0.03	0.05	0.03	0.05	0.06	0.04
2. Bakery3	0.70	0.25	0.69	0.25	∞	3.68
3. MutAst	1.46	0.37	1.00	0.37	0.22	0.59
4. Peterson	56.49	0.10	∞	0.10	∞	13.48
5. Ticket	∞	0.03	0.10	0.03	0.02	0.19
6. Berkeley RISC	0.01	0.04	$\bot$	0.04	0.01	0.02
7. DEC Firefly	0.01	0.02	$\bot$	0.03	0.01	0.07
8. IEEE Futurebus	0.26	0.68	$\bot$	$\bot$	∞	∞
9. Illinois University	0.01	0.03	$\bot$	0.03	∞	0.07
10. MESI	0.01	0.02	$\bot$	0.03	0.02	0.07
11. MOESI	0.01	0.06	$\bot$	0.05	0.02	0.08
12. Synapse N+1	0.01	0.02	$\bot$	0.02	0.01	0.01
13. Xerox PARC Dragon	0.01	0.05	$\bot$	0.06	0.02	0.10
14. Barber	0.62	0.21	$\bot$	0.21	∞	0.08
15. Bounded Buffer	0.01	3.10	0.01	3.16	∞	0.03
16. Unbounded Buffer	0.01	0.06	0.01	0.06	0.04	0.04
17. CSM	56.39	7.69	$\bot$	7.69	∞	125.32
18. Consistency	∞	0.11	$\bot$	0.11	∞	324.14
19. Insertion Sort	0.03	0.06	0.04	0.06	0.18	0.02
20. Selection Sort	∞	0.21	$\bot$	0.21	∞	0.33
21. Reset Petri Net	∞	0.02	$\bot$	$\bot$	∞	0.01
22. Train	42.24	59.21	$\bot$	$\bot$	∞	0.46
Number of verified properties	18	22	7	19	11	21

Table 1. Verification times (in seconds) using ALV [25]. '$\bot$' means termination with the answer 'Unable to verify' and '∞' means 'No answer' within 10 minutes.

The increase of the verification times in the Bounded Buffer example is due to the fact that the non-specialized system can easily be verified by a backward reachability analysis and, thus, the specialization we perform is unnecessary. Moreover, after specializing the Bounded Buffer system, we get a new system whose specification is quite large. We will return to this point in the next section.

5 Related Work and Conclusions

We have considered infinite state reactive systems specified by constraints over the integers and we have proposed a method, based on the specialization of CLP programs, for pre-processing the given systems and getting new, equivalent systems so that their backward (or forward) reachability analysis terminates with success more often (that is, precision is improved), without a significant increase of the verification time. The improvement of precision of the analysis is due to the fact that the backward (or forward) verification of the specialized systems takes into account the properties which are true on the initial states (or on the unsafe states, respectively).

The use of constraint logic programs in the area of system verification has been proposed by several authors (see [8,9], and [15] for a survey of early works).

Also transformation techniques for constraint logic programs have been shown to be useful for the verification of infinite state systems [12,13,21,23,24]. In the approach presented in this paper, constraint logic programs provide as an intermediate representation of the systems to be verified so that one can easily specialize those systems. To these constraint logic programs we apply a variant of the specialization technique presented in [13]. However, unlike [12,13,21,23,24], the final result of our specialization is not a constraint logic program, but a new reactive system which can be analyzed by using *any* verification tool for reactive systems specified by linear constraints on the integers. In this paper we have used the ALV tool [25] to perform the verification task on the specialized systems (see Section 4), but we could have also used (with minor syntactic modifications) other verification tools, such as TReX [2], FAST [3], and LASH [20]. Thus, one can apply to the specialized systems any of the optimization techniques implemented in those verification tools, such as *fixpoint acceleration*. We leave it for future research to evaluate the combined use of our specialization technique with other available optimization techniques.

Our specialization method is also related to some techniques for abstract interpretation [6] and, in particular, to those proposed in the field of verification of infinite state systems [1,5,7,16]. For instance, program specialization makes use of *generalization* operators [13] which are similar to the widening operators used in abstract interpretation. The main difference between program specialization and abstract interpretation is that, when applied to a given system specification, the former produces an *equivalent* specification, while the latter produces a more abstract (possibly, finite state) model whose semantics is an approximation of the semantics of the given specification. Moreover, since our specialization method returns a new system specification which is written in the same language of the given specification, after performing specialization we may also apply abstract interpretation techniques for proving system properties. Finding combinations of program specialization and abstract interpretation techniques that are most suitable for the verification of infinite state systems is an interesting issue for future research.

A further relevant issue we would like to address in the future is the reduction of the size of the specification of the specialized systems. Indeed, in one of the examples considered in Section 4, the time performance of the verification was not quite good, because the (specification of the) specialized system had a large size, due to the introduction of a large number of new predicate definitions. This problem can be tackled by using techniques for controlling *polyvariance* (that is, for reducing the number of specialized versions of the same predicate), which is an important issue studied in the field of program specialization [19].

Finally, we plan to extend our specialization technique to specifications of other classes of reactive systems such as *linear hybrid systems* [14,17].

Acknowledgements

This work has been partially supported by PRIN-MIUR, CNR (Italy), and CNRS (France). The last author has been supported by an ERCIM grant. Thanks to Laurent Fribourg and John Gallagher for many stimulating conversations.

References

1. P.A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction (On efficient verification of parameterized systems). *Int. J. of Foundations of Computer Science*, 20(5):779–801, 2009.
2. A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. *Proc. CAV'01*, LNCS 2102, 368–372. Springer, 2001.
3. 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.
4. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. POPL'77*, 238–252. ACM Press, 1977.
7. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19(2):253–291, 1997.
8. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 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. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
11. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
12. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proc. VCL'01*, Tech. Rep. DSSE-TR-2001-3, 85–96. Univ. of Southampton, UK, 2001.
13. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In *Proc. LOPSTR 2010*, LNCS 6564, 164–183. Springer, 2011.
14. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HYTECH. In *Proc. HSCC 2005*, LNCS 3414, 258–273. Springer, 2005.
15. L. Fribourg. Constraint logic programming applied to model checking. In *Proc. LOPSTR '99*, LNCS 1817, 31–42. Springer-Verlag, 2000.
16. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. CONCUR '01*, LNCS 2154, 426–440. Springer, 2001.
17. T. A. Henzinger. The theory of hybrid automata. In *Proc. LICS '96*, 278–292, 1996.
18. J. Jaffar and M. Maher. Constraint logic programming: A survey. *J. of Logic Programming*, 19/20:503–581, 1994.
19. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
20. LASH homepage: http://www.montefiore.ulg.ac.be/∼boigelot/research/lash.
21. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In *Proc. LOPSTR '99*, LNCS 1817, 63–82. Springer, 2000.
22. MAP homepage: http://www.iasi.cnr.it/∼proietti/system.html.
23. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *Proc. LOPSTR 2002*, LNCS 2664, 90–108, 2003.
24. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proc. TACAS 2000*, LNCS 1785, 172–187. Springer, 2000.
25. 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.

Appendix. Specialization Method for Forward Reachability

Let us briefly describe the *Fw-Specialization* method to be applied as a preprocessing step before performing a forward reachability analysis.

Fw-Specialization consists of three Steps (1f), (2f), and (3f), analogous to Steps (1), (2), and (3) of the backward reachability case described in Section 3.

Step (1f). **Translation**. Consider the system $Sys = \langle Var, Init, Trans\rangle$ and the property *Safe* specified as indicated in Step (1) of Section 3. The specification $\langle Sys, Safe\rangle$ is translated into the following constraint logic program *Fw* that encodes the forward reachability algorithm.

G_1: $unsafe \leftarrow u_1(X) \wedge fwReach(X)$
...
G_n: $unsafe \leftarrow u_n(X) \wedge fwReach(X)$
R_1: $fwReach(X') \leftarrow t_1(X, X') \wedge fwReach(X)$
...
R_m: $fwReach(X') \leftarrow t_m(X, X') \wedge fwReach(X)$
H_1: $fwReach(X) \leftarrow init_1(X)$
...
H_k: $fwReach(X) \leftarrow init_k(X)$

Note that we have interchanged the roles of the initial and unsafe states (compare the clauses G_i's and H_i's of program *Fw* with clauses I_i's and U_i's of program *Bw* presented in Section 3), and we have reversed the direction of the derivation of new states from old ones (compare clauses R_i's of program *Fw* with clauses T_i's of program *Bw*).

Step (2f). **Forward Specialization**. Program *Fw* is transformed into an equivalent program *SpFw* by applying a variant of the specialization algorithm described in Figure 1 to the input program *Fw*, instead of program *Bw*. This transformation consists in specializing *Fw* with respect to the disjunction *Unsafe* of constraints that characterizes the unsafe states of the system *Sys*.

Step (3f). **Reverse Translation**. The output of the specialization algorithm is a program *SpFw* of the form:

L_1: $unsafe \leftarrow u_1(X) \wedge newu_1(X)$
...
L_n: $unsafe \leftarrow u_n(X) \wedge newu_n(X)$
P_1: $newp_1(X') \leftarrow p_1(X, X') \wedge newd_1(X)$
...
P_r: $newp_r(X') \leftarrow p_r(X, X') \wedge newd_r(X)$
W_1: $newq_1(X) \leftarrow w_1(X)$
...
W_s: $newq_s(X) \leftarrow w_s(X)$

where (i) $p_1(X, X'), \ldots, p_r(X, X'), w_1(X), \ldots, w_s(X)$ are constraints, and (ii) the (possibly non-distinct) predicate symbols $newu_i$'s, $newp_i$'s, $newd_i$'s, and $newq_i$'s are the new predicate symbols introduced by the specialization algorithm.

Now we translate the program *SpFw* into a new specification $\langle SpSys, SpSafe\rangle$, where $SpSys = \langle SpVar, SpInit, SpTrans\rangle$. The translation is like the one presented in Step (3), the only difference being the interchange of the initial states and the unsafe states. In particular, (i) we derive a new variable declaration *SpVar* by introducing a new enumerated variable ranging over the set of new predicate symbols, (ii) we extract the disjunction *SpInit* of constraints characterizing the new initial states from the constrained facts W_i's, (iii) we extract the disjunction *SpTrans* of constraints characterizing the new transition relation from the clauses P_i's, (iv) we extract the disjunction *SpUnsafe* of constraints characterizing the new unsafe states from the clauses L_i's which define the *unsafe* predicate, and finally, (v) we define *SpSafe* as the formula $\neg\,\mathsf{EF}\,\mathit{SpUnsafe}$.

Similarly to Section 3, we can prove the correctness of the transformation consisting of Steps (1f), (2f), and (3f).

Theorem 5 (Correctness of Fw-Specialization). *Let $\langle SpSys, SpSafe\rangle$ be the specification derived by applying the Fw-Specialization method to the specification $\langle Sys, Safe\rangle$. Then, $\langle Sys, Safe\rangle$ is equivalent to $\langle SpSys, SpSafe\rangle$.*

Starting from the specification of Example 1, by applying our Fw-Specialization method, we get the following specialized specification:

SpVar: **enumerated** x_p $\{new1, new2\}$; **integer** x_1; **integer** x_2;
SpInit: $x_1 \geq 1 \wedge x_2 = 0 \wedge x_p = new2$;
SpTrans: $(x_1 < 1 \wedge x_p = new2 \wedge x'_1 = x_1 + x_2 \wedge x'_2 = x_2 + 1 \wedge x'_p = new1) \vee$
$(x_p = new2 \wedge x'_1 = x_1 + x_2 \wedge x'_2 = x_2 + 1 \wedge x'_p = new2)$
SpSafe: $\neg\,\mathsf{EF}(x_2 > x_1 \wedge x_p = new2)$

The forward reachability algorithm implemented in ALV successfully verifies the safety property of this specialized specification, while it is not able to verify (within 600 seconds) the safety property of the initial specification of Example 1.