Fundamenta Informaticae 119 (2012) [Revised June 19, 2013] 281–300
IOS Press

Improving Reachability Analysis of Infinite State Systems by Specialization

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

Alberto Pettorossi
Department of Informatics, Systems, and Production
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
Department of Informatics, Systems, and Production
University of Rome Tor Vergata
Via del Politecnico 1, 00133 Rome, Italy
senni@disp.uniroma2.it

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 the reachability analysis that works backwards from the unsafe states.

For reasons of efficiency, during specialization we make use of a relaxation from integers to reals. In particular, we test the satisfiability or entailment of constraints over the real numbers, while preserving the reachability properties of the reactive systems when constraints are interpreted over the integers.

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 a significant degradation of the time performance.

Keywords: Reachability analysis, automatic verification, program transformation, constraint logic programming.

1. Introduction

One of the present challenges in the field of automatic verification of reactive systems is the extension of the model checking techniques [7] 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 [15] 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 [6, 13, 14, 24, 26]. 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;
(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, 4, 23, 30, 40]. Some of them also use techniques borrowed from the field of *abstract interpretation* [9], 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, respectively) 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 $\overline{FR} \cap U \neq \emptyset$, respectively), one cannot conclude that there exists a state where φ does not hold.

One weakness of the Backward Strategy is that, when computing the set 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 excessively 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* [29] for mitigating 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) [27] 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.

The specialization process performed at Step 2 makes use of the familiar unfold/fold transformation rules for CLP programs [16]. For applying these rules many constraint manipulations and satisfiability tests are needed. Since constraint solving in the domain of the integers is often expensive, in order to improve efficiency, we apply a relaxation from integers to reals [5, 14], called the *real relaxation*, that is, we solve constraints over the reals $\mathbb{R}$, instead of the integers $\mathbb{Z}$. To this aim we reformulate the transformation rules so that the applicability conditions of the rules are based on the satisfiability or entailment of constraints over the reals. We prove that, even if we apply the real relaxation, every application of the rules transforms a given program into a new program with the same least model constructed over the integers, called the *least* $\mathbb{Z}$*-model*.

This relaxation allows us to exploit efficient techniques for checking satisfiability and entailment of constraints over the reals [38], for computing their projection, and for more complex constructions, such as the widening and the convex hull operations over sets of constraints [3, 9, 11]. However, since we use transformation rules which preserve the least $\mathbb{Z}$-model of the CLP programs, we may apply to the specialized programs other suitable verification techniques, including techniques based on constraints over the integers.

We have implemented our specialization method on the MAP transformation system for CLP programs [32] and we have performed experiments on several infinite state systems by using the *Action Language Verifier* (ALV) [40]. 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. Constraint Logic Programs over Integers

We will consider constraint logic programs with constraints over a finite domain and linear constraints over the set $\mathbb{Z}$ of the integer numbers.

Constraints are defined as follows. We assume a finite set $\mathbb{D}$ of typed constants. Variables can be either: (i) *integer* variables, or (ii) *enumerated* variables. Each integer variable ranges over the integer numbers and each enumerated variable X_i ranges over a finite set D_i of constants, with $D_i \subseteq \mathbb{D}$. The set D_i is said to be the *type* of the variable X_i and it is said to be the type of every constant in D_i. 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 variables and 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.

For the purposes of this paper it is sufficient to consider CLP programs which are finite sets of clauses of the form $A \leftarrow c \wedge B$, where A is an atom, c is a constraint, and B is either the empty conjunction *true* or a single atom. For reasons of simplicity and without loss of generality, we assume that the arguments of all atoms are variables, that is, the atoms are of the form $p(X_1, \ldots, X_n)$, with $n \geq 0$, where p is a predicate symbol not in $\{>, \geq, =, \neq\}$ and $X_1, \ldots, X_n$ are distinct integer or enumerated variables.

Given a constraint c, by *vars*(c) we denote the set of variables occurring in c. By $\forall(c)$ we denote the universal closure $\forall X_1 \ldots \forall X_n\, c$, where *vars*$(c) = \{X_1, \ldots, X_n\}$. Similarly, by $\exists(c)$ we denote the existential closure $\exists X_1 \ldots \exists X_n\, c$. Similar notation will also be used for atoms, goals, and clauses.

For constraints over the integers we assume the standard interpretation which interprets the $+$ and $\times$ symbols as addition and multiplication, respectively, and assigns to the $>, \geq, =$ symbols the usual meaning for integer comparison. Additionally, we adopt the standard Herbrand interpretation for constraints over $\mathbb{D}$, so that $e_1 = e_2$ holds if and only if e_1 and e_2 are identical constants.

By abuse of language, we will denote this constraint interpretation by $\mathbb{Z}$.

A $\mathbb{Z}$-*model* of a CLP program P is defined to be a model of P which agrees with the interpretation $\mathbb{Z}$ for the constraints. Every CLP program P has a unique *least* $\mathbb{Z}$-*model* (or, simply, *least model*), denoted $M(P)$ (see [27] for the definition of the least model of a constraint logic program).

We say that a constraint c is $\mathbb{Z}$-*satisfiable* if $\mathbb{Z} \models \exists(c)$. We also say that a constraint c $\mathbb{Z}$-*entails* a constraint d, denoted $c \sqsubseteq_{\mathbb{Z}} d$, if $\mathbb{Z} \models \forall(c \rightarrow d)$.

Let $\mathbb{R}$ be the usual interpretation of the constraints over the set of the real numbers, extended with the interpretation for constraints over $\mathbb{D}$ defined above. A constraint c is $\mathbb{R}$-*satisfiable* if $\mathbb{R} \models \exists(c)$. A constraint c $\mathbb{R}$-*entails* a constraint d, denoted $c \sqsubseteq_{\mathbb{R}} d$, if $\mathbb{R} \models \forall(c \rightarrow d)$. The $\mathbb{R}$-*projection* of a constraint c onto the set X of variables is a constraint c_p such that the following hold: (i) *vars*$(c_p) \subseteq X$ and (ii) $\mathbb{R} \models \forall(c_p \leftrightarrow \exists Y_1 \ldots \exists Y_k\, c)$, where $\{Y_1, \ldots, Y_k\} =$ *vars*$(c) - X$. Recall that the set of constraints over $\mathbb{Z}$ is not closed under projection.

The following lemma states some simple relationships between $\mathbb{Z}$-satisfiability and $\mathbb{R}$-satisfiability, and between $\mathbb{Z}$-entailment and $\mathbb{R}$-entailment.

Lemma 2.1. Let c and d be constraints and X be a set of variables.
(i) If c is $\mathbb{Z}$-satisfiable, then c is $\mathbb{R}$-satisfiable. (ii) If $c \sqsubseteq_{\mathbb{R}} d$, then $c \sqsubseteq_{\mathbb{Z}} d$. (iii) If c_p is the $\mathbb{R}$-projection of c onto X, then $c \sqsubseteq_{\mathbb{Z}} c_p$.

3. Transformation Rules with the Real Relaxation

In this section we present a set of transformation rules that can be used for specializing CLP programs. The applicability conditions of the rules are given in terms of constraints interpreted over the set $\mathbb{R}$ and, as shown by Theorem 3.1, these rules preserve the least $\mathbb{Z}$-model semantics.

The rules we will consider are those needed in the specialization step of the method presented in Section 5. Note, however, that the correctness result stated in Theorem 3.1 can be extended to constraint logic programs with any number of atoms in the body of the clauses (as done in [22]), with locally stratified negation [17], and to a larger set of rules, including *definition introduction* with $m\,(\geq 1)$ clauses, *negative unfolding*, *multiple positive folding* [19], and *negative folding* [19, 39].

Before presenting these rules, we show through an example that, if we consider different domains for the interpretation of the constraints and, in particular, if we apply the relaxation from the integers to the reals, we may derive different programs with different intended semantics.

Let us consider, for instance, the following constraint logic program P:

1. $p \leftarrow Y\!>\!0 \wedge Y\!<\!1$ 2. $q \leftarrow$

If we interpret the constraints over the reals, since $\mathbb{R} \models \exists Y(Y > 0 \wedge Y < 1)$, program P can be transformed into program $P_{\mathbb{R}}$:

$1'$. $p \leftarrow$ 2. $q \leftarrow$

If we interpret the constraints over the integers, since $\mathbb{Z} \models \neg\exists Y(Y > 0 \wedge Y < 1)$, program P can be transformed into program $P_{\mathbb{Z}}$:

2. $q \leftarrow$

Programs $P_{\mathbb{R}}$ and $P_{\mathbb{Z}}$ are not equivalent because they have different least $\mathbb{Z}$-models (which in this case coincide with their least Herbrand models). Thus, when we apply the real relaxation we should proceed with some care. In particular, we will admit a transformation rule only when its applicability conditions interpreted over $\mathbb{R}$ imply the corresponding applicability conditions interpreted over $\mathbb{Z}$.

The transformation rules are used to construct a *transformation sequence*, that is, a sequence $P_0, \ldots, P_n$ of programs. A transformation sequence $P_0, \ldots, P_n$ is incrementally constructed, starting from the initial program P_0, as we now indicate. Suppose that we have constructed a transformation sequence $P_0, \ldots, P_k$, for some k, with $0 \leq k \leq n-1$. The next program P_{k+1} in the transformation sequence is derived from program P_k by the application of a transformation rule among rules R1–R4 defined below.

Our first rule is the *Constrained Atomic Definition* rule (or *Definition Rule*, for short), which is applied for introducing a new predicate definition.

R1. Constrained Atomic Definition. Let us consider a clause, called a *definition clause*, of the form:

D: $newp(X_1, \ldots, X_h) \leftarrow c \wedge p(X_1, \ldots, X_h)$

where: (i) *newp* does not occur in $\{P_0, \ldots, P_k\}$, (ii) $X_1, \ldots, X_h$ are distinct variables, (iii) c is a constraint with $vars(c) \subseteq \{X_1, \ldots, X_h\}$, and (iv) p occurs in P_0.
By *constrained atomic definition* from program P_k we derive the program $P_{k+1} = P_k \cup \{D\}$. For $k \geq 0$, $Defs_k$ denotes the set of clauses introduced by the definition rule during the transformation sequence $P_0, \ldots, P_k$. In particular, $Defs_0 = \emptyset$.

R2. Unfolding. Let $E: \ H \leftarrow c \wedge A$ be a clause in program P_k and let

$K_1 \leftarrow c_1 \wedge B_1 \quad \ldots \quad K_m \leftarrow c_m \wedge B_m \qquad (m \geq 0)$

be all clauses of (a renamed apart variant of) program P_k such that, for $i\!=\!1, \ldots, m$, there exists a renaming substitution ρ_i such that $A = K_i\rho_i$ (recall that all atoms in a CLP program have distinct variables as arguments).
By *unfolding clause E w.r.t. the atom A* we derive the clauses

$F_1: \ H \leftarrow c \wedge c_1\rho_1 \wedge B_1\rho_1$
$\ldots$
$F_m: \ H \leftarrow c \wedge c_m\rho_m \wedge B_m\rho_m$

and from program P_k we derive the program $P_{k+1} = (P_k - \{E\}) \cup \{F_1, \ldots, F_m\}$.

Note that if $m=0$ then, by unfolding, clause E is deleted from P_k.

R3. Folding. Let E: $H \leftarrow c \wedge A$ be a clause in P_k and let D: $K \leftarrow d \wedge B$ be a clause in (a renamed apart variant of) *Defs*$_k$. Suppose that there exists a renaming substitution ρ such that: (i) $A = B\,\rho$, and (ii) $c \sqsubseteq_{\mathbb{R}} d\,\rho$. By *folding E using D* we derive the clause F: $H \leftarrow c \wedge K\rho$ and from program P_k we derive the program $P_{k+1} = (P_k - \{E\}) \cup \{F\}$.

The following example illustrates an application of Rule R3.

Example 3.1. Suppose that the following clause belongs to P_k:

E: $h(X) \leftarrow X \geq 1 \wedge 2\,Y = 3\,X + 2 \wedge p(X,Y)$

and suppose that the following clause is a definition clause in *Defs*$_k$:

D: $new(V,Z) \leftarrow Z > 2 \wedge p(V,Z)$

We have that the substitution $\rho = \{V/X, Z/Y\}$ satisfies Conditions (i) and (ii) of the folding rule because $X \geq 1 \wedge 2\,Y = 3\,X + 2 \sqsubseteq_{\mathbb{R}} (Z > 2)\rho$. Thus, by folding clause F using clause E, we derive:

F: $h(X) \leftarrow X \geq 1 \wedge 2\,Y = 3\,X + 2 \wedge new(X,Y)$

The following notion will be used for introducing the *clause removal* rule. Given two clauses of the form F: $H \leftarrow c \wedge B$ and E: $H \leftarrow d$, respectively, we say that F is $\mathbb{Z}$-*subsumed* by E, if $c \sqsubseteq_{\mathbb{Z}} d$. Similarly, we say that F is $\mathbb{R}$-*subsumed* by E, if $c \sqsubseteq_{\mathbb{R}} d$.

By Lemma 2.1, if γ is $\mathbb{R}$-subsumed by δ, then γ is $\mathbb{Z}$-subsumed by δ.

R4. Clause Removal. Let F be a clause in P_k of the form $H \leftarrow c \wedge B$. By *clause removal* we derive the program $P_{k+1} = P_k - \{F\}$ if either

Case (f): the constraint c is not $\mathbb{R}$-satisfiable, or

Case (s): clause F is $\mathbb{R}$-subsumed by a clause occurring in $P_k - \{F\}$.

The following Theorem 3.1 states that the transformation rules R1–R4 preserve the least $\mathbb{Z}$-model semantics.

Theorem 3.1. (Correctness of the Transformation Rules)
Let P_0 be a CLP program and let $P_0, \ldots, P_n$ be a transformation sequence obtained by applying rules R1–R4. Let us assume that for every k, with $0 < k < n-1$, if P_{k+1} is derived by applying folding to a clause in P_k using a clause D in $Defs_k$, then there exists j, with $0 < j < n-1$, such that: (i) D belongs to P_j, and (ii) P_{j+1} is derived by unfolding D w.r.t. the only atom in its body.
Then, for every ground atom A whose predicate occurs in P_0, we have that $A \in M(P_0)$ iff $A \in M(P_n)$.

Proof:
(Sketch) Let us consider variants of Rules R1–R4 where the applicability conditions are obtained from those for R1–R4 by replacing $\mathbb{R}$ by $\mathbb{Z}$. Let us denote R1$_{\mathbb{Z}}$–R4$_{\mathbb{Z}}$ these variants of the rules. Rules R1$_{\mathbb{Z}}$, R2$_{\mathbb{Z}}$, and R3$_{\mathbb{Z}}$, and R4$_{\mathbb{Z}}$ (Case (f) and Case (s)) can be viewed as instances (for $\mathcal{D} = \mathbb{Z}$) of the rules R1, R2p, R3 (Case P), R4f and R4s, respectively, for specializing CLP($\mathcal{D}$) programs presented in [17]. Since every constraint logic program without negation is trivially locally stratified, by Theorem 3.3.10 of [17] we have that $A \in M(P_0)$ iff $A \in M(P_n)$, for every ground atom A whose predicate occurs in P_0. By Lemma 2.1 we have that the applicability conditions of R1–R4 imply the applicability conditions of R1$_{\mathbb{Z}}$–R4$_{\mathbb{Z}}$, and, thus, we get the thesis. □

4. Specifying Reactive Systems

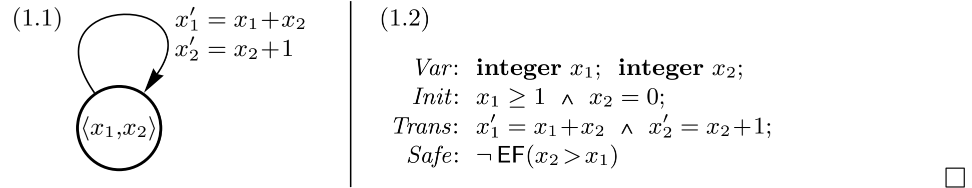
In order to specify reactive systems and their safety properties, we use a simplified version of the specification languages proposed in [2, 4, 30, 40].

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) enumerated or integer variables. An enumerated variable x is declared by the statement: **enumerated** x D, meaning that x ranges over a finite set D of constants. An integer variable x is declared by the statement: **integer** x. 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. *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 *safety specification* (or, for brevity, 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 4.1. On the left (1.1) we show a reactive system and on the right (1.2) we show the safety specification $\langle Sys, Safe\rangle$, where $Sys = \langle Var, Init, Trans\rangle$. In this specification the disjunction of constraints *Init* and *Trans* are both made out of a single constraint only.



□

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 the associated 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* is said to be an *initial* state. A state satisfying *Unsafe* is said to be an *unsafe* state.

A pair $\langle s, s'\rangle$ of states *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 $\neg\,\mathsf{EF}$*Unsafe*, if there is no state s which is reachable from an initial state and $s \models$ *Unsafe*.

A specification $\langle Sys_1, Safe_1\rangle$ is *equivalent* to a specification $\langle Sys_2, Safe_2\rangle$ if Sys_1 satisfies $Safe_1$ if and only if Sys_2 satisfies $Safe_2$.

5. Constraint-Based Specialization of Reactive Systems

Now we present a method for transforming a specification $\langle Sys, Safe\rangle$ 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, so that 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 symmetric [21] and it is described in the Appendix.) Bw-Specialization transforms the specification $\langle Sys, Safe\rangle$ into an equivalent specification $\langle SpSys, SpSafe\rangle$ according to the following three steps.

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

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

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

The specialized specification $\langle SpSys, SpSafe\rangle$ 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 in practice (see Section 6 for an experimental validation of this fact).

Now let us describe Steps (1), (2), and (3) in more detail.

Step (1). **Translation**. Let us consider the system $Sys = \langle Var, Init, Trans\rangle$ 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 $init_1(X) \vee \ldots \vee init_k(X)$ 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 $\neg\, \mathsf{EF}\, Unsafe$, 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: $unsafe \leftarrow init_1(X) \wedge bwReach(X)$
...
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 5.1. For the system specified in Example 4.1 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 translation of the specification $\langle$*Sys*, *Safe*$\rangle$ performed during Step (1) is correct in the sense stated by Theorem 5.1. 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 introduced in Section 4.

Theorem 5.1. (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 denoted by the formula *Init* and it does so by using the transformation rules with the relaxation from integers to reals introduced in Section 3.

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.

The correctness of the specialization algorithm directly follows from the correctness of the transformation rules stated in Theorem 3.1, whose applicability conditions are satisfied by the definition of the specialization algorithm itself.

In order to guarantee the termination of this Step (2), our specialization algorithm makes use also 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'). By Theorem 3.1 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, by Lemma 2.1, $e(X,X') \sqsubseteq_{\mathbb{R}} g(X')$ implies that $e(X,X') \sqsubseteq_{\mathbb{Z}} g(X')$.

The generalization operator, called *WidenSum*, we have used in our experiments reported in Section 6, 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 here in detail the operator *WidenSum* and, instead, we will refer to [22, 35]. In these papers the reader will also find the definition of various other generalization operators which can be used for specializing constraint logic programs. Here 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 [11], our generalization operator guarantees that during specialization only a finite number of new predicates is introduced.

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

INITIALIZATION (rules R1 and R3):
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 (rule R2):
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 (rule R4):
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 (rules R1 and R3):
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;

InDefs := *InDefs* $- \{C\}$;
SpBw := *SpBw* $\cup$ *SpC*;
end-while

Figure 1. The specialization algorithm.

Thus, we have the following result.

Theorem 5.2. (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 5.2. The following program is obtained as output of the specialization algorithm when it takes as input the CLP program of Example 5.1 and uses the generalization operator *WidenSum* [22]:

J_1: *unsafe* $\leftarrow x_1 \geq 1 \wedge x_2 = 0 \wedge$ *new*$1(x_1, x_2)$
S_1: *new*$1(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 = 0 \wedge x'_1 = x_1 \wedge x'_2 = 1 \wedge$ *new*$2(x'_1, x'_2)$
S_2: *new*$2(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 = 1 \wedge x'_1 = x_1 + 1 \wedge x'_2 = 2 \wedge$ *new*$3(x'_1, x'_2)$
S_3: *new*$3(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 \geq 1 \wedge x'_1 = x_1 + x_2 \wedge x'_2 = x_2 + 1 \wedge$ *new*$3(x'_1, x'_2)$
V_1: *new*$3(x_1, x_2) \leftarrow x_1 \geq 1 \wedge x_2 > x_1$ □

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 Rule R1 during the execution of the specialization algorithm of Step (2). Let *NewPred* be the set of all 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 variable declarations:

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

Example 5.3. The following specialized specification is the result of the reverse translation of the specialized CLP program of Example 5.2:

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: $\neg\, \mathsf{EF}(x_1 \geq 1 \wedge x_2 > x_1 \wedge x_p = new3)$

The backward reachability algorithm implemented in the ALV tool [40] is *not* able to verify the property *Safe* for the given system $Sys = \langle Var, Init, Trans\rangle$ of Example 4.1, while it is able to verify the property *SpSafe* for the system $SpSys = \langle SpVar, SpInit, SpTrans\rangle$ derived after specialization and defined as indicated above. (Note that, as a consequence of Theorem 5.4 below the specification $\langle Sys, Safe\rangle$ is equivalent to the specification $\langle SpSys, SpSafe\rangle$.)

This behaviour of ALV is basically due to the fact that for the given system, working backward from the unsafe states where $x_2 > x_1$ holds, ALV generates, at the various iterations, inequalities of the form: $x_1 + n\, x_2 + n\,(n-1)/2 \leq 0$, for $n = 1, 2, \ldots$ Thus, the verification process does not terminate.

Our specialization algorithm performs a forward exploration of the state space from the constraint $x_1 \geq 1 \wedge x_2 = 0$ that characterizes the initial states, and proves that the states which are reachable from the initial states all satisfy the constraint $x_1 \geq 1 \wedge x_2 \geq 0$. This fact is exploited for deriving from the given specification, a transformed specification (see, in particular, the constraints $x_1 \geq 1$, $x_2 = 0$, $x_2 = 1$, and $x_2 \geq 1$ that are inserted at various points of the specification shown above) and the constraint $x_1 \geq 1 \wedge x_2 \geq 0$ makes the inequalities $x_1 + n\, x_2 + n\,(n\text{–}1)/2 \leq 0$ to be unsatisfiable, for $n = 1, 2, \ldots$ Thus, for the transformed specification obtained after specialization, ALV does not generate these inequalities when performing the backward reachability analysis and it manages to verify the safety property. □

The correctness of our Bw-Specialization method is stated by the following theorem, which is a straightforward consequence of Theorems 5.1, 5.2, and 5.3.

Theorem 5.4. (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 $\langle SpSys, SpSafe\rangle$.

6. 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 [4, 13, 14, 40].

We have performed 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 [40]. 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 [32], 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, the following times expressed in seconds: (i) the time taken by ALV for verifying the given system (see columns *Sys*), and (ii) the total time taken by MAP for specializing the system plus the time taken by ALV for verifying the specialized system (see 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 for the given reactive system (regardless of the option used), but it verifies the property for 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. Unfortunately, in the Bounded Buffer example (with options 'default' and 'A') our specialization method significantly increases the verification time. All in all we may conclude that in the systems we have considered, the increase of precision due to our specialization method does not determine a significant degradation of the time performance.

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 backward reachability analysis and, thus, our pre-processing based on specialization is unnecessary. Another reason for this increase of the verification time in the Bounded Buffer example is that, after specialization, we get a new system whose specification is quite large (because the MAP system generates a large number of clauses). We will return to this point in the next section.

7. 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 [13, 14], and [24] 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 [18, 22, 31, 35, 37]. In the approach presented in this paper, constraint logic programs provide a suitable intermediate representation of the systems to be verified so that one can easily specialize those systems. To these

	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	$\perp$	0.04	0.01	0.02
7. DEC Firefly	0.01	0.02	$\perp$	0.03	0.01	0.07
8. IEEE Futurebus	0.26	0.68	$\perp$	$\perp$	∞	∞
9. Illinois University	0.01	0.03	$\perp$	0.03	∞	0.07
10. MESI	0.01	0.02	$\perp$	0.03	0.02	0.07
11. MOESI	0.01	0.06	$\perp$	0.05	0.02	0.08
12. Synapse N+1	0.01	0.02	$\perp$	0.02	0.01	0.01
13. Xerox PARC Dragon	0.01	0.05	$\perp$	0.06	0.02	0.10
14. Barber	0.62	0.21	$\perp$	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	$\perp$	7.69	∞	125.32
18. Consistency	∞	0.11	$\perp$	0.11	∞	324.14
19. Insertion Sort	0.03	0.06	0.04	0.06	0.18	0.02
20. Selection Sort	∞	0.21	$\perp$	0.21	∞	0.33
21. Reset Petri Net	∞	0.02	$\perp$	$\perp$	∞	0.01
22. Train	42.24	59.21	$\perp$	$\perp$	∞	0.46
Number of verified properties	18	22	7	19	11	21

Table 1. Verification times (in seconds) using ALV [40]. '$\perp$' means termination with the answer 'Unable to verify' and '∞' means 'No answer' within 10 minutes.

constraint logic programs we apply a variant of the specialization technique presented in [22]. However, unlike [18, 22, 31, 35, 37], 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 [40] to perform the verification task on the specialized systems (see Section 6), but we could have also used (with minor syntactic modifications) other verification tools, such as TReX [2], FAST [4], and LASH [30]. Thus, one can apply to the specialized systems any of the optimization techniques used by 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 [9] and, in particular, to those proposed in the field of verification of infinite state systems [1, 7, 12, 25]. For instance, program specialization makes use of *generalization* operators [22] 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 derives invariants by computing *approximations* of the semantics of the given specification. Similarly to abstract interpretation, also our specialization method derives invariants, which are encoded by the clauses introduced by using the definition rule. However, while abstract interpretation uses invariants for checking whether or not they are strong enough to prove a given system property, our method uses invariants for deriving equivalent specifications such that the verification task may terminate more often and, when it terminates, it may be made more efficient. 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 apply in a subsequent step some 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 related technique, which is widely used in the field of Software Model Checking, is CEGAR (*Counterexample-Guided Abstraction Refinement*) [28]. As already mentioned in the Introduction, when an upper approximation of the backward reachable states $\overline{BR}$ has a non-empty intersection with the set I of the initial states, that is, $\overline{BR} \cap I \neq \emptyset$, one has to verify whether this is due to a genuine counterexample violating the safety property or it is due to the approximation which is too coarse. In the latter case, by using a so-called interpolant formula, a smaller upper approximation of BR is produced so that the counterexample is avoided. This refinement process is repeated until the system is proved either safe or unsafe. Since the verification problem is undecidable, termination of the refinement process is not guaranteed. In contrast to abstraction refinement, our specialization method proceeds in the opposite direction and, starting from the original, concrete specification, it iteratively introduces progressively more abstract predicate definitions. The termination of our specialization method is guaranteed by the generalization operator that we use. Again, due to the undecidability of verification, the termination of the reachability analysis after specialization is not guaranteed.

Our Bw-Specialization method performs backward reachability after a program specialization phase, which can be regarded as a forward constraint propagation technique (and symmetrically for Fw-Specialization). Thus, our method is related to the combined use of backward and forward abstract interpretation, originally proposed by Cousot [8] and recently reconsidered and extended in [10, 36], where the result of backward analysis is used for performing forward analysis, and vice versa. However, while we have shown the effectiveness of our method in a number of examples of infinite state systems, no such extensive experimental evaluation is provided in [10, 36].

Another important feature of our method is that our transformation rules for CLP programs make use of the relaxation from the integers to the reals. The main practical advantage of using that relaxation is that we can then use the highly optimized tools for manipulating constraints over the reals, like, for example, PPL [3]. While relaxation techniques are typically used as *approximation* techniques (see, for instance, [5, 9, 11, 14, 35]), in this paper we have shown that some of them may realize transformations that preserve the semantics of interest. As an additional advantage, our approach allows one to use, after transformation, other analysis tools, like those working on integer constraints. Note, however, that in

principle, the result of applying transformation rules with the real relaxation may be sub-optimal with respect to the one which can be achieved by manipulating integer constraints. For example, using our rules we are unable to remove a clause whose constraints are satisfiable on the reals, but are unsatisfiable on the integers. However, we have checked that, for the significant set of examples of Section 6, this sub-optimality never occurs.

Since our specialization algorithm may introduce different specialized versions of the *bwReach*(X) predicate, it can be classified as a *polyvariant* specialization technique [29]. Polyvariant specialization is related to polyvariant program analysis, which has been studied in various fields, such as control flow analysis of functional programming languages (see [33] for a survey) and context-sensitive analysis of imperative programs [34]. A 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 6, the time performance of the verification was not quite good, because of the large size of the (specification of the) specialized system, due to the very many new predicate definitions that were introduced. 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 also studied in the field of program specialization. Some results in this direction have been presented in [20].

Finally, we plan to extend our specialization technique to specifications of other classes of reactive systems such as *linear hybrid systems* [23, 26].

Acknowledgements

This work has been partially supported by PRIN-MIUR and by a joint project between CNR (Italy) and CNRS (France). The last author has been supported by an ERCIM grant during his stay at LORIA-INRIA. Thanks to Laurent Fribourg and John Gallagher for many stimulating conversations about the topics of this paper. Many thanks to the anonymous referees for constructive comments.

References

[1] P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction (On efficient verification of parameterized systems). *International Journal of Foundations of Computer Science*, 20(5):779–801, 2009.

[2] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification, CAV 2001, Paris, France, July 18-22, 2001*, Lecture Notes in Computer Science 2102, pages 368–372. Springer, 2001.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[4] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008.

[5] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proceedings of CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24–27, 1999*, Lecture Notes in Computer Science 1664, pages 178–193. Springer, 1999.

[6] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 4th ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*, pages 238–252. ACM Press, 1977.

[10] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In G. Filé and H. Riis Nielson, editors, *Proceedings of the Fourteenth International Symposium on Static Analysis, SAS '07, Kongens Lyngby, Denmark*, Lecture Notes in Computer Science 4634, pages 333–348. Springer, 22–24 August 2007.

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

[12] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[13] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

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

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

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

[17] F. Fioravanti. *Transformation of Constraint Logic Programs for Software Specialization and Verification*. PhD thesis, Università di Roma "La Sapienza", Italy, 2002.

[18] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM SIGPLAN Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.

[19] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.

[20] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Controlling polyvariance for specialization-based verification. In *Proceedings of the 26th Italian Conference on Computational Logic, CILC 2011, 31 August – 2 September 2011, Pescara, Italy*, volume 810 urn:nbn:de:0074-810-0, pages 179–197. CEUR-WS, 2011.

[21] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. In G. Delzanno and I. Potapov, editors, *Proceedings of the 5th International Workshop on Reachability Problems, RP 2011, September 28-30, 2011, Genova, Italy*, Lecture Notes in Computer Science 6945, pages 165–179. Springer, 2011.

[22] 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 2012 doi:10.1017/S1471068411000627. Special Issue on the 25th GULP Annual Conference, 2012.

[23] 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.

[24] L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '99, Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 31–42. Springer-Verlag, 2000.

[25] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR '01*, Lecture Notes in Computer Science 2154, pages 426–440. Springer, 2001.

[26] T. A. Henzinger. The theory of hybrid automata. In *11th IEEE Symposium on Logic in Computer Science, LICS '96*, pages 278–292, 1996.

[27] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[28] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

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

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

[31] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '99, Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 2000.

[32] MAP. The MAP transformation system. Available from:
`http://www.iasi.cnr.it/∼proietti/system.html`, 1995–2010.

[33] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 2012. Forthcoming.

[34] H. R. Nielson, F. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[35] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17–20, 2002, Revised Selected Papers*, Lecture Notes in Computer Science 2664, pages 90–108, 2003.

[36] F. Ranzato, O. Rossi Doria, and F. Tapparo. A forward-backward abstraction refinement algorithm. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Proceedings 9th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI '08, San Francisco, CA, USA*, Lecture Notes in Computer Science 4905, pages 248–262. Springer, 2008.

[37] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Berlin, Germany*, Lecture Notes in Computer Science 1785, pages 172–187. Springer, 2000.

[38] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[39] H. Seki. On negative unfolding in the answer set semantics. In M. Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17–18, 2008, Revised Selected Papers*, Lecture Notes in Computer Science 5438, pages 168–184. Springer, 2009.

[40] 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.

8. Appendix. Specialization Method for Forward Reachability

Let us briefly describe the *Fw-Specialization* method which transforms a given specification into as equivalent one which is then subject to the 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 5.

Step (1f). **Translation**. Consider the system $Sys = \langle Var, Init, Trans \rangle$ and the property *Safe* specified as indicated in Step (1) of Section 5. 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 5), 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 presented in Figure 1 to the input program *Fw*, instead of the input 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) of Section 5, 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 $\neg\,\mathsf{EF}\,SpUnsafe$.

Similarly to Section 5, the following Theorem 8.1 shows the correctness of the transformation consisting of Steps (1f), (2f), and (3f).

Theorem 8.1. (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 4.1, by applying our Fw-Specialization method (with the generalization operator *CHWidenSum* [22]), 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: $\neg\,\mathsf{EF}(x_2 > x_1 \wedge x_p = new2)$

As for the backward reachability algorithm, also 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 4.1.