ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
"Antonio Ruberti"
CONSIGLIO NAZIONALE DELLE RICERCHE

F.Fioravanti, A. Pettorossi, M. Proietti, V. Senni

GENERALIZATION STRATEGIES FOR THE VERIFICATION OF INFINITE STATE SYSTEMS

R. 10-21, 2010

Fabio Fioravanti – Dipartimento di Scienze, Università 'G. D'Annunzio', Viale Pindaro 42, I-65127 Pescara, Italy. Email : fioravanti@sci.unich.it.
URL : http://fioravanti.sci.unich.it/fabio .

Alberto Pettorossi – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email : pettorossi@info.uniroma2.it. URL : http://www.iasi.cnr.it/~adp.

Maurizio Proietti – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email : maurizio.proietti@iasi.cnr.it.
URL : http://www.iasi.cnr.it/~proietti.

Valerio Senni – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy.
Email : senni@info.uniroma2.it. URL : http://www.disp.uniroma2.it/users/senni.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.cnr.it
URL: http://www.iasi.cnr.it

Abstract

We present a method for the automated verification of temporal properties of infinite state systems. Our verification method is based on the specialization of constraint logic programs (CLP) and works in two phases: (1) in the first phase, a CLP specification of an infinite state system is specialized with respect to the initial state of the system and the temporal property to be verified, and (2) in the second phase, the specialized program is evaluated by using a bottom-up strategy. The effectiveness of the method strongly depends on the generalization strategy which is applied during the program specialization phase. We consider several generalization strategies obtained by combining techniques already known in the field of program analysis and program transformation, and we also introduce some new strategies. Then, through many verification experiments, we evaluate the effectiveness of the generalization strategies we have considered. Finally, we compare the implementation of our specialization-based verification method to other constraint-based model checking tools. The experimental results show that our method is competitive with the methods used by those other tools.

KEYWORDS: Program Transformation, Program Verification, Constraint Logic Programming, Infinite State Reactive Systems.

1. Introduction

We consider the problem of verifying properties of reactive systems, that is, systems which continuously react to inputs by changing their internal state and producing outputs. One of the most challenging problems in this area is the extension of the model checking technique (see [10] for a thorough overview) from finite state systems to infinite state systems. In infinite state model checking the evolution over time of a system is modelled as a binary transition relation over an infinite set of states and the properties of that evolution are specified by means of propositional temporal formulas. In particular, in this paper we consider the *Computation Tree Logic* (CTL), which is a branching time propositional temporal logic by which one can specify, among others, the so-called *safety* and *liveness* properties [10].

Unfortunately, the verification of CTL formulas for infinite state systems is, in general, an undecidable problem. Thus, in order to cope with this limitation, various *decidable subclasses* of systems and formulas have been identified (see, for instance, [1, 17]). Other approaches to overcome the undecidability limitation are based on the enhancement of finite state model checking by using either more general *deductive techniques* (see, for instance, [43, 47]) or *abstractions*, by which one can compute conservative approximations of sets of states (see, for instance, [2, 7, 9, 12, 26, 27]).

Constraint logic programming (CLP) provides an excellent framework for specifying and verifying properties of reactive systems [23]. Indeed, the fixpoint semantics of logic programming languages allows us to easily represent the fixpoint semantics of various temporal logics [16, 40, 44] and constraints over the integers or the reals can be used to provide finite representations of infinite sets of states [16, 24].

However, for programs that specify infinite state systems, the proof procedures normally used in CLP, such as the extensions of SLDNF resolution and tabled resolution [8], very often diverge when trying to check some given temporal properties. This is due to the limited ability of these proof procedures to cope with infinitely failed derivations. For this reason, instead of using direct program evaluation, many logic programming-based verification systems make use of reasoning techniques such as: (i) *abstract interpretation* [4, 16], and (ii) *program transformation* [19, 36, 38, 41, 45]. In the techniques based on abstract interpretation one can construct approximations of the least and greatest fixpoints of (the immediate consequence operator associated with) a CLP program and then check the properties of interest on these approximations, while in the techniques based on program transformation one can pre-process the specification of a given system and a given property so that the verification itself becomes easier to perform.

This paper presents a verification method based on *program specialization*, a transformation technique that improves a program by exploiting the knowledge about the specific context where the program is used [25, 31, 35]. The verification method is an extension of the one first proposed in [19] and is applicable to specifications of CTL properties of infinite state systems encoded as CLP programs with locally stratified negation, where the constraints are linear inequations over the rationals. The verification method works in two phases. In Phase (1) we specialize the CLP programs with respect to the initial state of the systems and the temporal properties to be verified, and in Phase (2) we construct the perfect model of the specialized programs derived at the end of Phase (1), by applying a bottom-up evaluation procedure. As we will demonstrate through many examples below, this bottom-up procedure terminates in most cases without the need for abstractions.

The effectiveness of the verification method that we propose, strongly depends on the design of the *generalization strategy* which has to be applied during the program specialization phase.

Designing a good generalization strategy is not a trivial task: it must guarantee the termination of the specialization phase, and it should also provide a high precision and good performance. These requirements are often conflicting because, on the one hand, the use of a too coarse generalization strategy may determine the non-termination of Phase (2) and, thus, prevent the verification of many interesting properties and, on the other hand, a too specific generalization strategy may lead to verification times which are too long.

In this paper we introduce some new generalization strategies and we also propose various generalization strategies which are obtained by combining old techniques, already considered in the field of program analysis and program transformation (such as the *well-quasi orders* [33, 34, 37, 48] and the *convex hull* and *widening operators* [4, 11, 41]).

Our verification method has been implemented on the MAP transformation system [39]. We evaluate the effectiveness of this method by presenting the results of the experiments we have performed on several infinite state systems and temporal properties. We also compare the implementation of our verification method, with various constraint-based model checking tools and, in particular, with the following ones: (i) ALV [49], a system which combines BDD-based symbolic manipulation for boolean and enumerated types, with a solver for linear constraints on integers, (ii) DMC [16], which computes approximated least and greatest fixpoints of CLP(R) programs, and (iii) HyTech [29], a model checker for hybrid systems which handles constraints on reals. The experiments we have performed show that our method is effective and competitive with respect to the methods used by the other verification tools we have mentioned above.

The paper is structured as follows. In Section 2 we recall how CTL properties of infinite state systems can be encoded by using locally stratified CLP programs. In Section 3 we present our two-phase verification method. In Section 4 we describe various strategies that can be applied during the specialization phase and, in particular, the generalization strategies used for ensuring the termination of that first phase. In Section 5 we report on some experiments we have performed by using a prototype implemented on the MAP transformation system.

2. Specifying Reactive Systems and CTL Properties by CLP Programs

We assume familiarity with the basic notions of constraint logic programming [30] and model checking [10]. An infinite state system is modelled as a *Kripke structure*, denoted by a 4-tuple $\langle S, I, R, L\rangle$, where S is a set of *states*, $I \subseteq S$ is the set of *initial states*, R is a total binary *transition relation*, and L is a *labeling function* that associates with each state the set of *elementary properties* that hold in that state. A *computation path* in $\mathcal{K}$ is an *infinite* sequence of states $s_0\, s_1 \ldots$ such that, for every $i \geq 0$, $s_i\, R\, s_{i+1}$. The state s_{i+1} is called a *successor* of s_i. The properties to be verified will be specified as formulas of the *Computation Tree Logic* (CTL), whose syntax is:

$$\varphi ::= e \mid not(\varphi) \mid and(\varphi_1, \varphi_2) \mid ex(\varphi) \mid eu(\varphi_1, \varphi_2) \mid af(\varphi)$$

where e belongs to the set *Elem* of the elementary properties. Note that, in order to be consistent with the syntax of constraint logic programs, we slightly depart from the syntax of CTL given in [10].

The operators ex, eu, and af have the following semantics. The property $ex(\varphi)$ holds in a state s if there exists a successor s' of s such that φ holds in s'. The property $eu(\varphi_1, \varphi_2)$ holds in a state s if there exists a computation path π starting from s such that φ_1 holds in all states of a finite prefix of π and φ_2 holds in the first state of the rest of the path. The property $af(\varphi)$ holds in a state s if on every computation path π starting from s there exists a state s' where φ

holds. Formally, the semantics of CTL formulas is given by defining the satisfaction relation $\mathcal{K}, s \models \varphi$, which tells us when a formula φ holds in a state s of the Kripke structure $\mathcal{K}$ [10].

The set $\{ex, eu, af\}$ of operators is sufficient to express all CTL properties, because the other CTL operators can be defined in terms of ex, eu, and af. For instance: (i) the formula $ef(\varphi)$ (which holds in a state s iff there exists a computation path π starting from s and a state on π where φ holds) is defined as $eu(true, \varphi)$, and (ii) the formula $eg(\varphi)$ (which holds in a state s iff there exists a computation path π starting from s such that for every state on π the property φ holds) is defined as $not(af(true, \varphi))$.

In order to encode a Kripke structure and the satisfaction relation as a CLP program we will consider a set $\mathcal{C}$ of constraints and an interpretation $\mathcal{D}$ for the constraints in $\mathcal{C}$. We assume that: (i) $\mathcal{C}$ contains a set of *atomic constraints*, among which are *true*, *false*, and the equalities between terms, denoted by $t_1 = t_2$, (ii) $\mathcal{C}$ is closed under *conjunction* (denoted by comma), and (iii) $\mathcal{C}$ is closed under *projection*. The projection of a constraint c onto a tuple of variables X, denoted $project(c, X)$, is a constraint such that $\mathcal{D} \models \forall X \ (project(c, X) \leftrightarrow \exists Y c)$, where Y is the tuple of variables occurring in c and not in X. We also define a partial order $\sqsubseteq$ on $\mathcal{C}$ as follows: for any two constraints c_1 and c_2 in $\mathcal{C}$, we have that $c_1 \sqsubseteq c_2$ iff $\mathcal{D} \models \forall \ (c_1 \rightarrow c_2)$.

The semantics of a CLP program is defined as a *$\mathcal{D}$-model* [30], that is, a (possibly infinite) set of ground atoms whose truth implies the truth of all clauses of the program. Similarly to the case of logic programs, every *locally stratified* CLP program P has a unique *perfect $\mathcal{D}$-model* (also called *perfect model*, for short) which is denoted by $M(P)$. Recall that, for every locally stratified CLP program, the notions of perfect model, *stable model*, and *well-founded model* coincide [3].

Now, a Kripke structure $\langle S, I, R, L \rangle$ can be encoded as a CLP program as follows.
(1) A state in S is encoded as an n-tuple $\langle t_1, \ldots, t_n \rangle$ of terms. In what follows the variables X and Y are assumed to range over S.
(2) An initial state X in I is encoded as a clause of the form:

$$initial(X) \leftarrow c(X),$$

where $c(X)$ is a constraint.
(3) The transition relation R is encoded as a set of clauses of the form:

$$t(X, Y) \leftarrow c(X, Y)$$

where $c(X, Y)$ is a constraint. We also introduce a predicate *ts* such that, for every state X, Ys is a list of all the successor states of X iff $ts(X, Ys)$ holds, that is, for every state X, the state Y belongs to the list Ys iff $t(X, Y)$ holds. In [21] the reader will find: (i) an algorithm for deriving the clauses defining ts from the clauses defining t, and (ii) some conditions that guarantee that Ys is a finite list.
(4) The elementary properties which are associated with each state X by the labeling function L, are encoded by a set of clauses of the form:

$$elem(X, e) \leftarrow c(X)$$

where e is a constant, that is, the name of an elementary property, and $c(X)$ is a constraint.

The satisfaction relation $\models$ can be encoded by a predicate *sat* defined by the following clauses [19] (see also [38, 40] for similar encodings):

1. $sat(X, F) \leftarrow elem(X, F)$
2. $sat(X, not(F)) \leftarrow \neg sat(X, F)$
3. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1), sat(X, F_2)$
4. $sat(X, ex(F)) \leftarrow t(X, Y), sat(Y, F)$

5. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_2)$
6. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_1), t(X, Y), sat(Y, eu(F_1, F_2))$
7. $sat(X, af(F)) \leftarrow sat(X, F)$
8. $sat(X, af(F)) \leftarrow ts(X, Ys), sat\_all(Ys, af(F))$
9. $sat\_all([\,], F) \leftarrow$
10. $sat\_all([X|Xs], F) \leftarrow sat(X, F), sat\_all(Xs, F)$

Let $P_{\mathcal{K}}$ denote the constraint logic program consisting of clauses 1–10 together with the clauses defining the predicates *initial*, t, ts, and *elem*. We have that $P_{\mathcal{K}}$ is locally stratified and, thus, it has a unique perfect model $M(P_{\mathcal{K}})$.

Suppose that we want to verify that a CTL formula φ holds for all initial states. In order to do so we define a new predicate *prop* as follows:

$prop \equiv_{def} \forall X(initial(X) \rightarrow sat(X, \varphi))$

This definition can be encoded by the following two clauses:

γ_1 : $prop \leftarrow \neg negprop$
γ_2 : $negprop \leftarrow initial(X), sat(X, not(\varphi))$

The correctness of this encoding is stated by the following Theorem 2.1 (its proof can be found in [21]). Note that program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is locally stratified and it has a unique perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$.

Theorem 2.1 (Correctness of Encoding) *Let $\mathcal{K}$ be a Kripke structure, let I be the set of initial states of $\mathcal{K}$, and let φ be a CTL formula. Then,*

for all states $s \in I$, $\mathcal{K}, s \models \varphi$ iff $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$.

Example 1. Let us consider the reactive system depicted in Figure 1, where a state $\langle X_1, X_2 \rangle$, which is a pair of rationals, is denoted by the term $s(X_1, X_2)$.

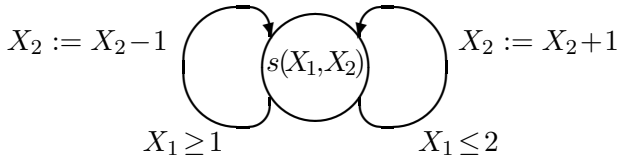

Figure 1: A reactive system. In any initial state we have that $X_1 \leq 0$ and $X_2 = 0$. The transitions do not change the value of X_1.

The Kripke structure $\mathcal{K}$ which models that system, is defined as follows. The initial states are given by the clause:

11. $initial(s(X_1, X_2)) \leftarrow X_1 \leq 0,\ X_2 = 0$

The transition relation R is given by the clauses:

12. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \geq 1,\ Y_1 = X_1,\ Y_2 = X_2 - 1$
13. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \leq 2,\ Y_1 = X_1,\ Y_2 = X_2 + 1$

The predicate ts is given by the clauses:

14. $ts(s(X_1, X_2), [s(Y_1, Y_2)]) \leftarrow X_1 < 1,\ Y_1 = X_1,\ Y_2 = X_2 + 1$
15. $ts(s(X_1, X_2), [s(Y_{11}, Y_{21}), s(Y_{12}, Y_{22})]) \leftarrow X_1 \geq 1,\ X_1 \leq 2,$
$Y_{11} = X_1,\ Y_{21} = X_2 - 1,\ Y_{12} = X_1,\ Y_{22} = X_2 + 1$

16. $ts(s(X_1, X_2), [s(Y_1, Y_2)]) \leftarrow X_1 > 2,\ Y_1 = X_1,\ Y_2 = X_2 - 1$

The elementary property *negative* is given by the clause:

17. $elem(s(X_1, X_2), negative) \leftarrow X_2 < 0$

Suppose that we want to verify the property that in every initial state $s(X_1, X_2)$, where $X_1 \leq 0$ and $X_2 = 0$, the CTL formula $not(eu(true, negative))$ holds, that is, from any initial state it is impossible to reach a state $s(X_1', X_2')$ where $X_2' < 0$. By using the fact that every CTL formula of the form $not(not(\varphi))$ is equivalent to φ, this property is encoded by the following two clauses:

γ_1: $prop \leftarrow \neg negprop$

γ_2: $negprop \leftarrow initial(X),\ sat(X, eu(true, negative))$

Note that, in this example, for the verification of *prop* the clauses defining the predicate $sat(X, af(F))$ (that is, clauses 7 and 8 of program $P_{\mathcal{K}}$) are not needed. Thus, clauses 14, 15, and 16, which define the predicate *ts*, are not needed either. □

Our encoding of the Kripke structure can easily be extended to provide witnesses of the formulas of the form $eu(\varphi_1, \varphi_2)$ and counterexamples of the formulas of the form $af(\varphi)$, as usual for model checkers of finite state systems [10]. Indeed, in order to do so, it is sufficient to add to the predicate *sat* an extra argument that recalls the sequence of states (or transitions) constructed during the verification of a given formula. For details, the reader may refer to [21].

3. Verifying Infinite State Systems by Specializing CLP Programs

In this section we present a method for checking whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is a CLP encoding of an infinite state system and *prop* is a predicate encoding the satisfiability of a given CTL formula.

As already mentioned, the proof procedures normally used in constraint logic programming, such as the extensions to CLP of SLDNF resolution and tabled resolution, very often diverge when trying to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ by evaluating the query *prop*. This is due to the limited ability of these proof procedures to cope with infinite failure.

Also the bottom-up construction of the perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ often diverges, because it does not take into account the information about the query *prop* to be evaluated, the initial states of the system, and the formula to be verified. Indeed, by a naive bottom-up evaluation, the clauses of $P_{\mathcal{K}}$ may generate infinitely many atoms of the form $sat(s, \psi)$. For instance, given a state s_0, an elementary property f that holds in s_0, and an infinite sequence $\{s_i \mid i \in \mathbb{N}\}$ of distinct states such that, for every $i \in \mathbb{N}$, $t(s_{i+1}, s_i)$ holds, clauses 5 and 6 generate by bottom-up evaluation the infinitely many atoms $sat(s_i, eu(true, f))$, for every $i \in \mathbb{N}$, and the infinitely many atoms: (i) $sat(s_0, f)$, (ii) $sat(s_0, eu(true, f))$, (iii) $sat(s_0, eu(true, eu(true, f)))$, ...

In this paper we will show that the termination of the bottom-up construction of the perfect model can be improved by a prior application of program specialization. In particular, in this section we will present our verification algorithm which consists of two phases: Phase (1), in which we specialize the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ with respect to the query *prop*, thereby deriving a new program P_S whose perfect model $M(P_S)$, also denoted M_S, satisfies the following equivalence: $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_S$, and Phase (2), in which we construct M_S by a bottom-up evaluation.

The specialization phase modifies the initial program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ by incorporating into the specialized program P_S the information about the initial states and the formula to be verified. The bottom-up evaluation of P_S may terminate more often than the bottom-up evaluation of

the initial program because: (i) it avoids the generation of an infinite set of states that are unreachable from the initial states, and (ii) it generates only specialized atoms corresponding to the subformulas of the formula to be verified.

The Verification Algorithm
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.
Output: The perfect model M_s of a CLP program P_S such that:
$prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_s$.
(Phase 1) $Specialize(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}, P_s)$;
(Phase 2) $BottomUp(P_s, M_s)$

The *Specialize* procedure of Phase (1) consists in the iterated application of two subsidiary procedures: (i) the *Unfold* procedure, which applies the *unfolding* rule and the *clause removal* rule, and (ii) the *Generalize&Fold* procedure, which applies the *definition introduction* rule and the *folding* rule. These program transformation rules are variants of the usual rules considered in the case of logic programs and constraint logic programs (see, for instance, [18, 20, 46]. These variants have been designed to easily meet the objective of performing program specialization.

Procedure *Specialize*
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.
Output: A stratified program P_S such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M(P_S)$.

$P_S := \{\gamma_1\}$; $InDefs := \{\gamma_2\}$; $Defs := \{\}$;
while there exists a clause γ in $InDefs$
do $Unfold(\gamma, \Gamma)$;
$Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$;
$P_S := P_S \cup \Phi$;
$InDefs := (InDefs - \{\gamma\}) \cup NewDefs$; $Defs := Defs \cup NewDefs$;
end-while

The *Unfold* procedure takes as input a clause $\gamma \in InDefs$ and returns as output a set Γ of clauses derived from clause γ by one or more applications of the unfolding rule. A single application of this rule is encoded by the *UnfoldOnce* function defined below. In the definition of the *UnfoldOnce* function we use the following notation. Given two atoms A and B, we denote by $A = B$ the constraint: (i) $t_1 = u_1, \ldots, t_n = u_n$, if A is of the form $p(t_1, \ldots, t_n)$ and B is of the form $p(u_1, \ldots, u_n)$, for some n-ary predicate symbol p, and (ii) *false*, otherwise.

Function $UnfoldOnce(\gamma, A)$
Let γ be a clause of the form $H \leftarrow c, Q, A, R$, where A is an atom whose predicate is defined in $P_{\mathcal{K}}$. Let $(K_1 \leftarrow c_1, B_1), \ldots, (K_m \leftarrow c_m, B_m)$, with $m \geq 0$, be all (renamed apart) clauses in program $P_{\mathcal{K}}$ such that, for $i = 1, \ldots, m$, the constraint $(c,\ A = K_i,\ c_i)$ is satisfiable.

$UnfoldOnce(\gamma, A) =$
$\{(K \leftarrow c, A = K_1, c_1, Q, B_1, R), \ldots, (K \leftarrow c, A = K_m, c_m, Q, B_m, R)\}$

At the first application of the *Unfold* procedure, the input clause γ is the clause $\gamma_2 :\ negprop \leftarrow initial(X),\ sat(X, not(\varphi))$, where $initial(X)$ and φ are the atoms that encode the initial states and the formula to be verified, respectively. The effect of the application of the *Unfold* procedure

is the propagation of the information about the initial states and the property to be verified through the Kripke structure represented by $P_{\mathcal{K}}$.

Procedure *Unfold*(γ, Γ)
Input: A clause γ in *InDefs*.
Output: A set Γ of clauses.

UNFOLD:
Let A be any atom in the body of γ;
$\Gamma :=$ *UnfoldOnce*(γ, A);
while there exist a clause δ in Γ and an atom A in the body of δ, such that A is of one of the following forms: (i) *initial*(s), (ii) $t(s_1, s_2)$, (iii) $ts(s, ss)$, (iv) *elem*(s, e), (v) *sat*(s, e), where e is an elementary property, (vi) *sat*$(s, not(\psi_1))$, (vii) *sat*$(s, and(\psi_1, \psi_2))$, (viii) *sat*$(s, ex(\psi_1))$, (ix) *sat_all*(ss, ψ_1), where ss is a non-variable list
do $\Gamma := (\Gamma - \{\delta\}) \cup$ *UnfoldOnce*(δ, A)
end-while;

REMOVE SUBSUMED CLAUSES:
while in Γ there exist two distinct clauses δ: $H \leftarrow c$ and η: $H \leftarrow d, G$ such that $d \sqsubseteq c$ (that is, η is subsumed by δ)
do $\Gamma := \Gamma - \{\eta\}$
end-while

Due to the structure of the clauses in $P_{\mathcal{K}}$, the *Unfold* procedure terminates for every $\gamma \in$ *InDefs*. In particular, in order to enforce termination, every atom of the form *sat*$(s, eu(\psi_1, \psi_2))$ or *sat*$(s, af(\psi_1))$ is selected at most once during each application of the procedure.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and introduces a set *NewDefs* of *definitions*, that is, clauses each of which is of the form δ: *newp*$(X) \leftarrow d(X)$, *sat*(X, ψ), where *newp* is a new predicate. Any such clause δ represents a set of states X satisfying the constraint $d(X)$ and the CTL property ψ, and incorporates the information propagated by the *Unfold* procedure about the initial state and property to be verified. All definitions introduced by the *Generalize&Fold* procedure are stored in a set *Defs* and can be used for folding during the current or the future applications of the procedure itself. By folding the clauses in Γ using the definitions in *Defs* $\cup$ *NewDefs*, the procedure derives a new set Φ of clauses which are added to the program P_S.

In the clauses of P_S derived upon termination of the *Specialize* procedure, there is no reference to the predicates used in $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, except for *prop* and *negprop*, that is, P_S provides a definition of *prop* and *negprop* in terms of the new predicates introduced by the applications of the *Generalize&Fold* procedure.

Unfortunately, an uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions and, therefore, it may make the *Specialize* procedure not to terminate. In order to guarantee termination, the *Generalize&Fold* procedure may introduce new definitions which are *more general* than definitions introduced by previous applications of the procedure, where a definition *newq*$(X) \leftarrow g(X)$, *sat*(X, ψ) is more general than *newp*$(X) \leftarrow b(X)$, *sat*(X, ψ) if $b(X) \sqsubseteq g(X)$. Thus, more general definitions correspond to larger sets of states.

In order to introduce generalized definitions in a suitable way, we will extend to constraint logic programs some techniques which have been proposed for controlling generalization in *pos-*

itive supercompilation [48] and *partial deduction* [33, 37]. The details of the *Generalize&Fold* procedure and the results stating the correctness and the termination of the *Specialize* procedure will be given in the next section.

In order to compute the perfect model $M_{\mathcal{S}}$ of $P_{\mathcal{S}}$ it is convenient to represent sets of ground atoms by sets of *facts*, that is, sets of (possibly non-ground) clauses of the form $H \leftarrow c$, where H is an atom and c is a constraint. A fact $H \leftarrow c$ represents the set of all the ground instances of H that satisfy c. Clearly, the perfect model of a program may have several different representations as a set of facts. The *BottomUp* procedure constructs a non-ground representation of $M_{\mathcal{S}}$ by using the *non-ground immediate consequence operator* $S_{P_{\mathcal{S}}}$, instead of the usual immediate consequence operator $T_{P_{\mathcal{S}}}$ [30]. Since program $P_{\mathcal{S}}$ is *stratified* (see in Theorem 4.4 below), in order to construct $M_{\mathcal{S}}$, the *BottomUp* procedure considers the strata of $P_{\mathcal{S}}$, starting from the lowest stratum and going up to the highest stratum. For each stratum the *BottomUp* procedure has to compute the least fixpoint of the restriction of $S_{P_{\mathcal{S}}}$ to that stratum. Since this fixpoint may be represented by an infinite set of facts, the *BottomUp* procedure may not terminate, although there is only a finite number of strata in $P_{\mathcal{S}}$. In Section 5 we will see that the *BottomUp* procedure, applied after the *Specialize* procedure, terminates in many significant cases and, upon termination, computes a finite non-ground representation of $M_{\mathcal{S}}$.

Example 2. Let us consider the reactive system $\mathcal{K}$ of Example 1. We want to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where *prop* expresses the fact that the formula $not(eu(true, negative))$ holds in every initial state. Now we have that: (i) by using a traditional Prolog system, the evaluation of the query *prop* does not terminate in the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ because *negprop* has an infinitely failed SLD tree, (ii) by using the XSB tabled logic programming system, the query *prop* does not terminate because infinitely many *sat* atoms are tabled, and (iii) the bottom-up construction of $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ does not terminate because of the presence of clauses 5 and 6 as we have indicated at the beginning of this section.

By applying the *Specialize* procedure to the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ (with a suitable generalization strategy, as illustrated in the next section), we derive the following specialized program $P_{\mathcal{S}}$:

γ_1. $prop \leftarrow \neg negprop$
γ_2'. $negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ new1(X_1, X_2)$
γ_3. $new1(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 = 0,\ Y_1 = X_1,\ Y_2 = 1,\ new2(Y_1, Y_2)$
γ_4. $new2(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 \geq 0,\ Y_1 = X_1,\ Y_2 = X_2 + 1,\ new2(Y_1, Y_2)$

Note that the *Specialize* procedure has propagated through the program $P_{\mathcal{S}}$ the constraint $X_1 \leq 0, X_2 = 0$ characterizing the initial states (see clause 11 of Example 1). This constraint, in fact, occurs in clause γ_3 and its generalization $X_1 \leq 0, X_2 \geq 0$ occurs in clause γ_4. The *BottomUp* procedure computes the perfect model of $P_{\mathcal{S}}$, which is $M_{\mathcal{S}} = \{prop\}$, in a finite number of steps (indeed, starting from the lowest stratum, we have that, for all X_1, X_2, $new2(X_1,X_2)$, $new1(X_1,X_2)$, and *negprop* are all false). Thus, the property $not(eu(true, negative))$ holds in every initial state of $\mathcal{K}$. □

4. Generalization Strategies

The design of a powerful generalization strategy should meet two conflicting requirements. The strategy, in fact, should introduce new definitions which are (i) as general as possible, so as to enforce the termination of the *Specialize* procedure, and (ii) as specific as possible, so as to retain the maximum information about the initial state and the property to be verified, and

produce a program P_S for which the *BottomUp* procedure terminates. In this section we present several generalization strategies for coping with those conflicting requirements. These strategies combine various by now standard techniques used in the fields of program transformation and static analysis, such as *well-binary relations*, *well-quasi orderings*, *widening*, and *convex hull* operators, and variants thereof [4, 11, 33, 34, 37, 41, 48]. All these strategies guarantee the termination of the *Specialize* procedure. However, since in general the verification problem is undecidable, the power and effectiveness of the different generalization strategies can only be assessed by performing experiments. The results of those experiments will be presented in the next section.

4.1. The *Generalize&Fold* Procedure

The *Generalize&Fold* procedure makes use of a tree of definitions, called *Definition Tree*, whose nodes are labelled by the clauses in $Defs \cup \{\gamma_2\}$. By construction there is a bijection between the set of nodes of the Definition Tree and $Defs \cup \{\gamma_2\}$ and, thus, we will identify each node with its label. The root of the Definition Tree is labelled by clause γ_2 (recall that $\{\gamma_2\}$ is the initial value of *InDefs*) and the children of a clause γ in $Defs \cup \{\gamma_2\}$ are the clauses *NewDefs* derived after applying the procedures $Unfold(\gamma, \Gamma)$ and $Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$. Our *Generalize&Fold* procedure is based on the combined use of a *firing relation* and a *generalization operator*. The firing relation determines *when* to generalize, while the generalization operator determines *how* to generalize.

Definition 4.1 (Well-Binary Relation $\lhd$ and Well-Quasi Ordering $\preceq$) A *well-binary relation* on a set S is a binary relation $\lhd$ such that, for every infinite sequence $e_0, e_1, \ldots$ of elements of S, there exist i and j such that $i < j$ and $e_i \lhd e_j$. A *well-quasi ordering* (or *wqo*, for short) on a set S is a reflexive, transitive, well-binary relation $\preceq$ on S. Given e_1 and e_2 in S, we write $e_1 \approx e_2$ if $e_1 \preceq e_2$ and $e_2 \preceq e_1$. We say that a wqo $\preceq$ is *thin* iff for all $e \in S$, the set $\{e' \in S \mid e \approx e'\}$ is finite.

Definition 4.2 (Firing Relation) *A firing relation is a well-binary relation on the set $\mathcal{C}$ of constraints.*

The firing relation guarantees that generalization is eventually applied and, thus, its role is similar to the one of the *whistle* algorithm [48].

Definition 4.3 (Generalization Operator $\ominus$) Let $\preceq$ be a thin wqo on the set $\mathcal{C}$ of constraints. A binary operator $\ominus$ on $\mathcal{C}$ is a *generalization operator* with respect to $\preceq$ if, for all constraints c and d in $\mathcal{C}$, we have: (i) $d \sqsubseteq c \ominus d$, and (ii) $c \ominus d \preceq c$. (Note that, in general, $\ominus$ is not commutative.)

The use of a thin wqo in Definition 4.3 guarantees that during the *Specialize* procedure each definition can be generalized a finite number of times only and, thus, the termination of the procedure is guaranteed. Definition 4.3 generalizes several operators proposed in the literature, such as the *most specific generalization* operator [37, 48] and the widening operator [11].

Procedure *Generalize&Fold*
Input: (i) a set *Defs* of definitions, (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure, (iii) a firing relation $\lhd$, and (iv) a generalization operator $\ominus$.

Output: (i) A set *NewDefs* of new definitions, and (ii) a set Φ of folded clauses.

NewDefs $:= \emptyset$; $\Phi := \Gamma$;
while in Φ there exists a clause η: $H \leftarrow e, G_1, L, G_2$, where L is either $sat(X, \psi)$ or $\neg sat(X, \psi)$ *do*

GENERALIZE:
Let $e_p(X)$ be $project(e, X)$.
1. *if* in *Defs* there exists a clause δ: $newp(X) \leftarrow d(X), sat(X, \psi)$ such that $e_p(X) \sqsubseteq d(X)$ (modulo variable renaming)
   then *NewDefs* := *NewDefs*
2. *elseif* there exists a clause α in *Defs* such that:
   (i) α is of the form $newq(X) \leftarrow b(X), sat(X, \psi)$, and (ii) α is the most recent ancestor of γ in the Definition Tree such that $b(X) \lhd e_p(X)$
   then *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow b(X) \ominus e_p(X), sat(X, \psi)\}$
3. *else* *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow e_p(X), sat(X, \psi)\}$

FOLD:
$\Phi := (\Phi - \{\eta\}) \cup \{H \leftarrow e, G_1, M, G_2\}$, where M is $newp(X)$, if L is $sat(X, \psi)$, and M is $\neg newp(X)$, if L is $\neg sat(X, \psi)$

end-while

The following theorem establishes that the *Specialize* procedure always terminates and preserves the perfect model semantics. The proof of this theorem (which is a simple variant of the proof of Theorem 3 in [21]), in order to show that the set of new definitions introduced during the *Specialize* procedure is finite, relies on the hypothesis that the firing relation $\lhd$ is a well-binary relation and the relation $\precsim$ used in the definition of the generalization operator $\ominus$, is a *thin* wqo.

Theorem 4.4 (Termination and Correctness of the *Specialize* Procedure) *(i) For every input program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, for every firing relation $\lhd$, and for every generalization operator $\ominus$, the Specialize procedure terminates. (ii) Let $P_{\mathcal{S}}$ be the output program of the Specialize procedure. Then (ii.1) $P_{\mathcal{S}}$ is stratified (and thus, locally stratified), and (ii.2) prop $\in M(P_{\mathcal{K}})$ iff prop $\in M(P_{\mathcal{S}})$.*

4.2. Firing Relations and Generalization Operators on Linear Constraints

In our verification experiments we will consider the set Lin_k of constraints defined as follows. Every constraint $c \in Lin_k$ is the conjunction of $m\,(\geq 0)$ *distinct* atomic constraints $a_1, \ldots, a_m$ and, for $i = 1, \ldots, m$, (1) a_i is either of the form $p_i \leq 0$ or of the form $p_i < 0$, and (2) p_i is a polynomial of the form $q_0 + q_1 X_1 + \ldots + q_k X_k$, where $X_1, \ldots, X_k$ are distinct variables and $q_0, q_1, \ldots, q_k$ are integer coefficients. An equation $r = s$ stands for the conjunction of the two inequations $r \leq s$ and $s \leq r$. In the sequel, when we write $c =_{def} a_1, \ldots, a_m$ we mean that the a_i's are the atomic constraints of c. The constraints in Lin_k are interpreted over the rationals in the usual way.

4.2.1. Firing Relations

Now we present four firing relations on the set Lin_k. These firing relations are called *Always*, *Maxcoeff*, *Sumcoeff*, and *Homeocoeff*. They are all wqo's.

a_1	a_2	$a_1 \precsim_A a_2$	$a_1 \precsim_M a_2$	$a_1 \precsim_S a_2$	$a_1 \precsim_H a_2$
$1-2X_1<0$	$3+X_1<0$	yes	yes	yes	yes
$2-2X_1+X_2<0$	$1+3X_1<0$	yes	yes	no	no
$1+3X_1<0$	$2-2X_1+X_2<0$	yes	no	yes	no

Table 1: Examples of firing relations $\precsim_A$, $\precsim_M$, $\precsim_S$, and $\precsim_H$.

(F1) The wqo *Always*, denoted by $\precsim_A$, is the relation $Lin_k \times Lin_k$.

(F2) The wqo *Maxcoeff*, denoted by $\precsim_M$, compares the maximum absolute values of the coefficients occurring in the polynomials. It is defined as follows. For any atomic constraint a of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, we define $maxcoeff(a) = \max\{|q_0|, |q_1|, \ldots, |q_k|\}$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_M a_2$ iff $maxcoeff(a_1) \leq maxcoeff(a_2)$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_M c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_M b_j$.

(F3) The wqo *Sumcoeff*, denoted by $\precsim_S$, compares the sum of the absolute values of the coefficients occurring in the polynomials. It is defined as follows. For any atomic constraint a of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, we define $sumcoeff(a) = \sum_{j=0}^{k} |q_j|$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_S a_2$ iff $sumcoeff(a_1) \leq sumcoeff(a_2)$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_S c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_S b_j$.

(F4) The wqo *Homeocoeff*, denoted by $\precsim_H$, compares sequences of absolute values of coefficients occurring in polynomials. It is an adaptation to Lin_k of the *homeomorphic embedding* operator [33, 34, 37, 48]. The wqo $\precsim_H$ takes into account the commutativity and the associativity of addition and conjunction and it is defined as follows. Given two polynomials $p_1 =_{def} q_0 + q_1X_1 + \ldots + q_kX_k$, and $p_2 =_{def} r_0 + r_1X_1 + \ldots + r_kX_k$, we have that $p_1 \precsim_H p_2$ iff there exists a permutation $\langle \ell_0, \ldots, \ell_k \rangle$ of the indexes $\langle 0, \ldots, k \rangle$ such that, for $i = 0, \ldots, k$, $|q_i| \leq |r_{\ell_i}|$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_H a_2$ iff $p_1 \precsim_H p_2$. Similarly, if we are given the atomic constraints $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$ we have that $c_1 \precsim_H c_2$ iff there exist m *distinct* indexes $\ell_1, \ldots, \ell_m$, with $m \leq n$, such that $a_i \precsim_H b_{\ell_i}$, for $i = 1, \ldots, m$.

Figure 2(A) illustrates the containment relationships between the firing relations *Always*, *Maxcoeff*, *Sumcoeff*, and *Homeocoeff*. Note that a generalization operator is applied less often if it is associated with a smaller firing relation. Figure 2(B) will be explained later.

Table 1 provides some examples of the firing relations and, in particular, it shows that the relations *Maxcoeff* and *Sumcoeff* are not comparable.

4.2.2. Generalization Operators

Now we present some generalization operators on Lin_k which we will use in the verification examples of the next section. For defining these operators we will use the relations $\precsim_M$, $\precsim_S$,

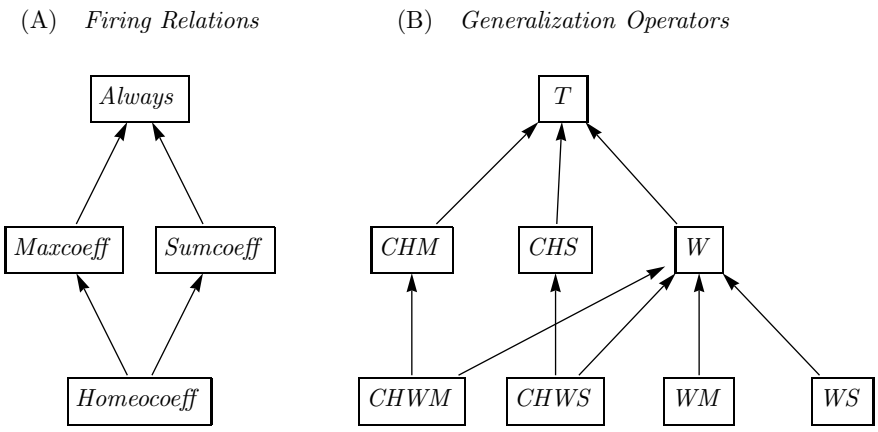

Figure 2: Firing relations and generalization operators. (A) An arrow $p \rightarrow q$ from firing relation p to firing relation q means $p \subseteq q$. (B) An arrow $g \rightarrow h$ from generalization operator g to generalization operator h means $\ominus_g \sqsubseteq \ominus_h$.

and $\precsim_H$, which are thin wqo's on Lin_k. On the contrary, the wqo $\precsim_A$ is *not* thin and it cannot be used for defining generalization operators.

(G1) Given any two constraints c and d, the operator *Top*, denoted $\ominus_T$, returns *true*. It can be shown that *Top* is a generalization operator with respect to any of the thin wqo's $\precsim_M$, $\precsim_S$, and $\precsim_H$. Since the *Top* operator forgets all information about its operands, it often performs an over-generalization and produces poorly specialized programs (see the experimental evaluation in Section 5).

(G2) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and d, the operator *Widen*, denoted $\ominus_W$, returns the constraint $a_{i1}, \ldots, a_{ir}$, such that $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$. Thus, *Widen* returns all atomic constraints of c that are entailed by d (see [11] for a similar widening operator used in static program analysis). The operator $\ominus_W$ is a generalization operator w.r.t. any of the thin wqo's $\precsim_M$, $\precsim_S$, and $\precsim_H$.

(G3) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and $d =_{def} b_1, \ldots, b_n$, the operator *WidenMax*, denoted $\ominus_{WM}$, returns the conjunction $a_{i1}, \ldots, a_{ir}, b_{j1}, \ldots, b_{js}$, where: (i) $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$, and (ii) $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim_M c\}$. The operator *WidenSum*, denoted $\ominus_{WS}$, is defined like *WidenMax*, with $\precsim_M$ replaced by $\precsim_S$. The operators $\ominus_{WM}$ and $\ominus_{WS}$ are generalization operators w.r.t. the thin wqo's $\precsim_M$ and $\precsim_S$, respectively.

The operators *WidenMax* and *WidenSum* are similar to *Widen* but, together with the atomic constraints of c that are entailed by d, they also return the conjunction of a subset of the atomic constraints of d. Note that the operator *WidenHomeo*, denoted $\ominus_{WH}$, which is defined like *WidenMax*, with $\precsim_M$ replaced by $\precsim_H$, is *not* a generalization operator w.r.t. $\precsim_H$. Indeed, the constraint $c \ominus_{WH} d$ may contain more atomic constraints than c and, thus, it may not be the case that $(c \ominus_{WH} d) \precsim_H c$.

Next we define some generalization operators by using the *convex hull* operator, which sometimes is used in the static program analysis [11]. The *convex hull* of two constraints c and d in Lin_k, denoted by $ch(c, d)$, is the least (w.r.t. the $\sqsubseteq$ ordering) constraint h in Lin_k such that $c \sqsubseteq h$ and $d \sqsubseteq h$. (Note that $ch(c, d)$ is unique up to equivalence of constraints.)

(G4) Given any two constraints c and d, let $ch(c, d)$ be of the form $b_1, \ldots, b_n$. The oper-

c	$-X_1 \leq 0,\ -2+X_1 \leq 0$	$1-X_1 \leq 0,\ -2+X_1 \leq 0$	$1-X_1 \leq 0, -1+X_1 \leq 0$ $X_2 \leq 0,\ -X_2 \leq 0$
d	$2-X_1 \leq 0,\ 1-X_2 \leq 0$	$-X_1 \leq 0$	$X_1 \leq 0,\ -X_1 \leq 0$ $2-X_2 \leq 0,\ -2+X_2 \leq 0$
$c \ominus_W d$	$-X_1 \leq 0$	*true*	$-1+X_1 \leq 0, -X_2 \leq 0$
$c \ominus_{WM} d$	$2-X_1 \leq 0,\ 1-X_2 \leq 0$	$-X_1 \leq 0$	$1-X_1 \leq 0,\ -1+X_1 \leq 0$ $-X_2 \leq 0$
$c \ominus_{CHM} d$	$-X_1 \leq 0$	$-X_1 \leq 0$	$-X_2 \leq 0$
$c \ominus_{CHWM} d$	$-X_1 \leq 0$	$-X_1 \leq 0$	$-1+X_1 \leq 0, -X_2 \leq 0$

Table 2: Examples of application of generalization operators.

ator *CHMax*, denoted $\ominus_{CHM}$, returns the conjunction $b_{j1}, \ldots, b_{js}$, such that $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim_M c\}$. The operator *CHSum*, denoted $\ominus_{CHS}$, is defined like *CHMax*, with $\precsim_M$ replaced by $\precsim_S$. The operators $\ominus_{CHM}$ and $\ominus_{CHS}$ are generalization operators w.r.t. the thin wqo's $\precsim_M$ and $\precsim_S$, respectively.

Both *CHMax* and *CHSum* return the conjunction of a subset of the atomic constraints of $ch(c, d)$. Note that an operator similar to *CHMax* which uses $\precsim_H$, instead of $\precsim_M$, is not a generalization operator.

(G5) Given any two constraints c and d, we define the operator *CHWidenMax*, denoted $\ominus_{CHWM}$, as follows: $c \ominus_{CHWM} d =_{def} c \ominus_{WM} ch(c, d)$. Similarly, we define the operator *CHWidenSum*, denoted $\ominus_{CHWS}$, as follows: $c \ominus_{CHWS} d =_{def} c \ominus_{WS} ch(c, d)$. The operators $\ominus_{CHWM}$ and $\ominus_{CHWS}$ are generalization operators w.r.t. the thin wqo's $\precsim_M$ and $\precsim_S$, respectively.

Both *CHWidenMax* and *CHWidenSum* return the conjunction of a subset of the atomic constraints of c and a subset of the atomic constraints of $ch(c, d)$.

Note that some other combinations of the widening and convex hull operators would not yield new generalization operators. Indeed, for all constraints c and d, we have that: (i) $c \ominus_T ch(c, d) = c \ominus_T d$, (ii) $c \ominus_W ch(c, d) = c \ominus_W d$, (iii) $c \ominus_{CHM} ch(c, d) = c \ominus_{CHM} d$, and (iv) $c \ominus_{CHS} ch(c, d) = c \ominus_{CHS} d$.

It can be shown that the generalization operators defined at points (G1)–(G5) above are pairwise distinct. Table 2 shows some examples of application of generalization operators.

In order to compare our generalization operators we extend the $\sqsubseteq$ partial ordering on constraints to a partial ordering, also denoted $\sqsubseteq$, on generalization operators, as follows: $\ominus_1 \sqsubseteq \ominus_2$ (that is, $\ominus_1$ is less general than $\ominus_2$) iff, for all constraints c and d, $(c \ominus_1 d) \sqsubseteq (c \ominus_2 d)$. Figure 2(B) shows the relationships between generalization operators. The operators not connected by any sequence of arrows are not comparable w.r.t. $\sqsubseteq$.

5. Experimental Evaluation

In this section we present the results of the experiments we have performed on several examples of verification of infinite state reactive systems. We have implemented the verification algorithm presented in Section 2 using MAP [39], an experimental system for transforming constraint logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `clpq` library

to operate on constraints.

We have considered the following *mutual exclusion* protocols and we have verified some of their properties. (i) *Bakery* [16]: we have verified safety (that is, mutual exclusion) and liveness (that is, starvation freedom) in the case of two processes, and safety in the case of three processes; (ii) *MutAst* [32]: we have verified safety in the case of two processes; (iii) *Peterson* [42]: we have verified safety in the case of $N\,(\geq 2)$ processes by considering a *counting abstraction* of the protocol [13]; and (iv) *Ticket* [16]: we have verified safety and liveness in the case of two processes.

We have also verified safety properties of the following *cache coherence* protocols: (v) *Berkeley RISC*, (vi) *DEC Firefly*, (vii) *IEEE Futurebus+*, (viii) *Illinois University*, (ix) *MESI*, (x) *MOESI*, (xi) *Synapse N+1*, and (xii) *Xerox PARC Dragon*. We have considered *parameterized* versions of the protocols (v)–(xii), that is, protocols designed for an arbitrary number of processors. We have applied our verification method to the counting abstractions described in [13].

Then we have verified safety properties of the following systems. (xiii) *Barber* [6]: we have considered a parameterized version of this protocol with a single barber process and an arbitrary number of customer processes; (xiv) *Bounded Buffer* and *Unbounded Buffer*: we have considered protocols for two producers and two consumers which communicate via a bounded and an unbounded buffer, respectively (the encodings of these protocols are taken from [16]); (xv) *Consprodjava*, which is (a counting abstraction of) a producer–consumer Java program realized using threads: we have verified that for any number of threads there is no deadlock [5]; (xvi) *CSM* is a central server model described in [15]; (xvii) *Consistency*, which is a directory-based consistency protocol for client–server distributed systems (proposed by Steven German) [5, 14]: we have considered two versions of the system and we have verified that mutual exclusion is preserved for any number of processes; (xviii) *Insertion Sort* and *Selection Sort*: we have considered the problem of checking array bounds of these two sorting algorithms, parameterized w.r.t. the size of the array, as presented in [16]; (xix) *Office Light Control* [49] is a protocol for controlling how office lights are switched on and off, depending on room occupancy; (xx) *Reset Petri Net* is a Petri Net augmented with *reset arcs*: we have considered a reachability problem for a net which is a variant of one presented in [36]; (xxi) *Kanban* is a Petri Net modelling a concurrent production system [5]: we have verified that the value of certain control variables are bound within some specified limits; (xxii) *Train* is an encoding of a control system for speed regulation of subway trains [5, 28]: we have verified that a train is never too early or too late with respect to its expected arrival time.

Tables 3 and 4 show the results of running the MAP system on the above examples by using the firing relation *Always* in conjunction with each of the eight generalization operators introduced in Section 4. In particular, Table 3 reports, for each example, the total verification time, that is, the time taken by the *Verification* algorithm, if it terminates, and Table 4 reports the *specialization time*, that is, the time taken by the *Specialize* procedure only. For a meaningful comparison between total specialization times, we have omitted from Table 4 the times relative to the *Consprodjava* example, for which the *Specialize* procedure does not terminate when using some of the generalization operators.

Let us compare the various generalization operators with respect to *precision*, that is, with respect to the number of properties verified. As expected, we have that precision increases when we use less general generalization operators, that is, precision is anti-monotonic with respect to the $\sqsubseteq$ relation (precision increases when going down in Figure 3(B)). This anti-monotonicity is explained by the fact that the use of less general generalization operators may produce specialized programs that better exploit the information about both the initial state

	T	W	CHM	CHS	$CHWM$	$CHWS$	WM	WS
Bakery2 (safety)	80	150	30	30	40	50	30	20
Bakery2 (liveness)	∞	∞	110	100	130	120	90	60
Bakery3 (safety)	3940	4900	430	430	460	460	170	170
MutAst	2330	320	390	400	420	440	80	160
Peterson	∞	∞	860	870	1380	1410	190	220
Ticket (safety)	20	30	30	20	20	10	20	20
Ticket (liveness)	110	100	100	100	80	90	80	110
Berkeley RISC	30	30	180	170	210	200	30	30
DEC Firefly	70	130	200	130	310	320	20	30
IEEE Futurebus+	16380	47570	∞	47120	75860	47630	110	2460
Illinois University	100	70	50	60	50	50	10	30
MESI	70	40	250	250	130	130	30	30
MOESI	100	150	330	170	120	170	40	60
Synapse N+1	10	20	10	30	30	30	20	20
Xerox PARC Dragon	50	60	220	220	280	270	30	30
Barber	∞	28440	2000	2050	2530	2560	1160	1220
Bounded Buffer	30	360	9490	9570	5800	5840	3580	3580
Unbounded Buffer	∞	∞	410	400	420	420	3810	3810
Consprodjava	∞	∞	∞	∞	∞	∞	25300	∞
CSM	∞	∞	3820	3880	4830	4860	6410	6540
Consistency v1	∞	∞	410	450	780	780	70	60
Consistency v2	∞	70	110	130	220	260	40	60
Insertion Sort	80	70	130	120	160	170	100	90
Selection Sort	∞	∞	∞	160	230	180	∞	180
Office Light Control	50	40	50	50	50	50	50	50
Reset Petri Net	∞	∞	∞	∞	∞	∞	20	20
Kanban	∞	∞	15630	15800	17790	18040	8130	8000
Train	∞	1440	3420	6290	3680	6650	30900	57260

Table 3: Verification times for the MAP system. For each example we show the *total verification time* (Phases 1 and 2) obtained by using the firing relation *Always* in conjunction with the generalization operators: $\ominus_T$, $\ominus_W$, $\ominus_{CHM}$, $\ominus_{CHS}$, $\ominus_{CHWM}$, $\ominus_{CHWS}$, $\ominus_{WM}$, and $\ominus_{WS}$. Times are expressed in milliseconds (ms). The precision is 10 ms. '0' means termination in less than 10 ms. '∞' means no answer within 100 seconds.

	T	W	CHM	CHS	$CHWM$	$CHWS$	WM	WS
Bakery2 (safety)	20	60	30	30	40	50	30	20
Bakery2 (liveness)	80	120	80	70	90	80	60	30
Bakery3 (safety)	690	610	410	410	440	440	160	160
MutAst	210	280	360	370	400	420	70	140
Peterson	20	250	850	870	1370	1400	190	220
Ticket (safety)	20	30	30	20	20	10	20	20
Ticket (liveness)	70	60	60	60	40	50	40	70
Berkeley RISC	20	20	150	140	180	170	30	30
DEC Firefly	20	60	100	70	150	160	20	30
IEEE Futurebus+	30	230	1540	300	1110	290	110	250
Illinois University	30	50	40	50	50	50	10	30
MESI	20	30	150	150	120	120	30	30
MOESI	30	60	140	80	100	80	40	50
Synapse N+1	10	10	10	20	30	20	20	10
Xerox PARC Dragon	20	30	190	190	260	250	30	30
Barber	400	1590	1870	1920	2400	2430	1130	1190
Bounded Buffer	10	140	9480	9560	5790	5830	2070	2070
Unbounded Buffer	20	100	410	400	410	410	350	350
CSM	30	450	3810	3870	4820	4840	6350	6480
Consistency v1	20	90	410	450	770	770	70	60
Consistency v2	20	70	110	130	220	260	30	50
Insertion Sort	20	50	130	120	150	160	100	90
Selection Sort	30	70	190	160	220	180	780	170
Office Light Control	40	30	40	40	40	40	40	40
Reset Petri Net	10	10	10	10	10	10	10	10
Kanban	20	1000	15590	15750	17740	18010	8070	7940
Train	20	50	3370	6230	3630	6590	4280	7560
TOTAL	1930	5550	39560	41470	40600	43120	24140	27130

Table 4: Specialization times for the MAP systems. For each example we show the *specialization time* (Phases 1 only) obtained by using the firing relation *Always* in conjunction with the generalization operators: $\ominus_T$, $\ominus_W$, $\ominus_{CHM}$, $\ominus_{CHS}$, $\ominus_{CHWM}$, $\ominus_{CHWS}$, $\ominus_{WM}$, and $\ominus_{WS}$. Times are expressed in milliseconds (ms). The precision is 10 ms. '0' means termination in less than 10 ms. '∞' means no answer within 100 seconds.

and the property to be verified.

Let us now compare the various generalization operators with respect to the specialization time. We have that specialization times increase when we use less general generalization operators, that is, specialization time is anti-monotonic with respect to the $\sqsubseteq$ relation (specialization time increases when going down in Figure 3(B)). This is due to the fact that less general generalization operators may introduce more definitions and, therefore, the specialization phase may take more time. Note also that the generalization operators that use the convex hull operators (that is, $\ominus_{CHM}$, $\ominus_{CHS}$, $\ominus_{CHWM}$, and $\ominus_{CHWS}$) exhibit higher specialization times than the ones that do not. This is due to the extra cost of computing the convex hull which, however, does not always correspond to an increase of precision.

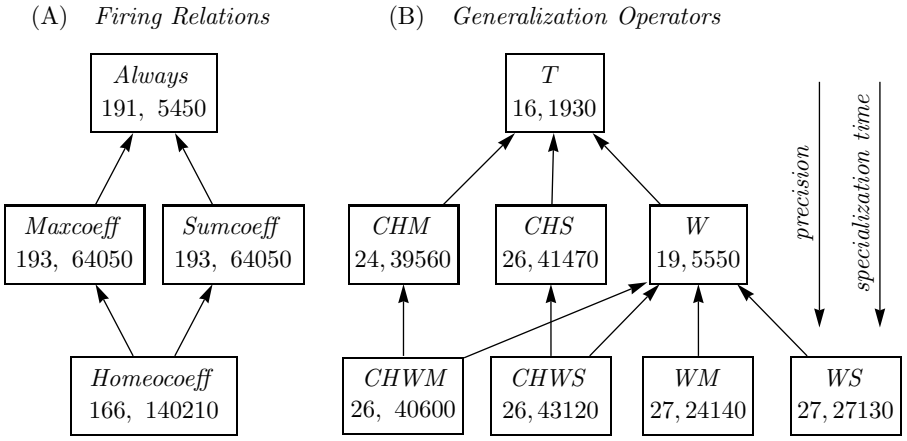

Figure 3: Comparison of firing relations and generalization operators. (A) For each firing relation the numbers on the left and on the right indicate, respectively: (i) the number of properties verified by using that firing relation in conjunction with all generalization operators, and (ii) the sum of the specialization times taken by using that firing relation in conjunction with all generalization operators. (B) For each generalization operator the numbers on the left and on the right indicate, respectively: (i) the total number of properties verified (see Table 3), and (ii) the sum of the specialization times (see Table 4).

If we compare the various generalization operators by using them in conjunction with each firing relation *Maxcoeff*, *Sumcoeff*, and *Homeocoeff*, instead of *Always*, we get similar anti-monotonicity results (not shown here) for precision and specialization times.

Let us now compare the firing relations *Always*, *Maxcoeff*, *Sumcoeff*, and *Homeocoeff*. We may expect that a firing relation that determines fewer generalization steps, also determines the introduction of more definitions and, therefore, we may expect that both precision and specialization time are anti-monotonic with respect to $\subseteq$ (they increase when going down in Figure 2(A)). This anti-monotonicity is, in fact, observed in our experiments except for the case of the *Homeocoeff* firing relation (see Figure 3(A)). This is explained by the fact that the specialization times obtained by using the *Homeocoeff* firing relation are very high and, therefore, the execution of the *Specialize* procedure is often longer than the time limit of 100 seconds we have assumed for the time out. Note also that the modest increase of precision from *Always* to *Maxcoeff* or *Sumcoeff* (from 191 to 193) is paid by a considerable increase of specialization time (from 5450 ms to 64050 ms).

In summary, if we consider the balance between precision and time, the generalization strate-

gies that use *Always* as firing relation and either $\ominus_{WM}$ or $\ominus_{WS}$ as generalization operators outperform all the others. In particular, the generalization strategies based on the homeomorphic embedding as a firing relation (that is, *Homeocoeff*) and the convex hull operator (that is, $\ominus_{CHM}$, $\ominus_{CHS}$, $\ominus_{CHWM}$, and $\ominus_{CHWS}$) turn out not to be the best strategies in our examples.

In order to compare the implementation of our verification method using MAP with other constraint-based model checking tools for infinite state systems available in the literature, we have done the verification examples described in Table 3 on the following systems as well: (i) ALV [49], which combines BDD-based symbolic manipulation for boolean and enumerated types, with a solver for linear constraints on integers, (ii) DMC [16], which computes (approximated) least and greatest fixpoints of CLP(R) programs, and (iii) HyTech [29], a model checker for hybrid systems which handles constraints on reals. All experiments with the MAP, ALV, DMC, and HyTech systems have been performed on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system. Table 5 reports the results obtained by using various options available in those verification systems.

Table 5 indicates that, in terms of precision, MAP with either the *WM* or the *WS* generalization operator is the best system (27 properties verified out of 28), followed by ALV with the *default* option and DMC with the *A* (abstraction) option (19 out of 28 for both systems), and HyTech with the *Bw* (backward reachability) option (18 out of 28).

In order to compare the systems in terms of verification times, now we consider the options that give the best precision, that is, MAP with *WM*, ALV with *default*, DMC with *A*, and HyTech with *Bw*. Then we compare MAP to each other system by computing the average verification time over the set of examples where both systems terminate. We have that MAP has better average time than ALV (2462 ms and 8033 ms average time, respectively, over the 19 examples where both systems terminate), and MAP has also better average time than DMC (95 ms and 928 ms, respectively, over 19 examples). However, MAP has a slightly worse average time than HyTech (519 ms and 331 ms, respectively, over 18 examples). This is explained by the fact that HyTech with the *Bw* option tries to verify a safety property with a very simple strategy, that is, by constructing the reachability set backwards from the property to be verified, while MAP applies much more sophisticated techniques. Note also that the average verification times are affected by the peculiar behaviour on some specific examples. For instance, in the Bounded Buffer and the Barber examples the MAP system has longer verification times with respect to HyTech, because these examples can be easily verified by backward reachability, and this makes it redundant the MAP specialization phase, which propagates the information about the initial state. On the opposite side, MAP is more efficient than HyTech in the IEEE Futurebus+ and Bakery 3 examples.

6. Conclusions

This paper extends earlier work presented in [19, 22].

We have presented a specialization-based method for the verification of CTL properties of infinite state reactive systems. Our method consists of two phases: in Phase (1) a CLP specification of the reactive system is specialized w.r.t. the initial state and the temporal property to be verified, and in Phase (2) the perfect model of the specialized program is constructed in a bottom-up way.

For Phase (1) we have focused on the generalization strategy which is applied during program specialization and which often determines the quality of the specialized program. We have considered various generalization strategies that employ different firing relations, for deciding

EXAMPLE	MAP		ALV				DMC		HyTech	
	WM	*WS*	*default*	*A*	*F*	*L*	*noAbs*	*Abs*	*Fw*	*Bw*
Bakery 2 (safety)	30	20	20	30	90	30	10	30	∞	20
Bakery 2 (liveness)	90	60	30	30	90	30	60	70	$\times$	$\times$
Bakery 3 (safety)	170	170	580	570	∞	600	460	3090	∞	360
MutAst	80	160	$\perp$	$\perp$	910	$\perp$	150	1370	70	130
Peterson N	190	220	71690	$\perp$	∞	∞	∞	∞	70	∞
Ticket (safety)	20	20	∞	80	30	∞	∞	60	∞	∞
Ticket (liveness)	80	110	∞	230	40	∞	∞	220	$\times$	$\times$
Berkeley RISC	30	30	10	$\perp$	20	60	30	30	∞	20
DEC Firefly	20	20	10	$\perp$	20	80	50	80	∞	20
IEEE Futurebus+	110	2460	320	$\perp$	∞	670	4670	9890	∞	380
Illinois University	10	30	10	$\perp$	∞	140	70	110	∞	20
MESI	30	30	10	$\perp$	20	60	40	60	∞	20
MOESI	40	60	10	$\perp$	40	100	50	90	∞	10
Synapse N+1	20	20	10	$\perp$	10	30	0	0	∞	0
Xerox PARC Dragon	30	30	20	$\perp$	40	340	70	120	∞	20
Barber	1160	1220	340	$\perp$	90	360	140	230	∞	90
Bounded Buffer	3580	3580	0	10	∞	20	20	30	∞	10
Unbounded Buffer	3810	3810	10	10	40	40	∞	∞	∞	20
Consprodjava	25300	∞	∞	∞	∞	∞	∞	∞	∞	∞
CSM	6410	6540	79490	$\perp$	∞	∞	∞	∞	∞	∞
Consistency v1	70	60	∞	$\perp$	∞	∞	∞	∞	∞	2030
Consistency v2	40	60	∞	$\perp$	40	∞	∞	∞	∞	2790
Insertion Sort	100	90	40	60	∞	70	30	80	∞	10
Selection Sort	∞	180	∞	390	∞	∞	∞	∞	∞	∞
Office Light Control	50	50	20	20	30	20	10	10	∞	∞
Reset Petri Nets	20	20	∞	$\perp$	∞	10	0	0	∞	10
Kanban	8130	8000	∞	∞	∞	∞	∞	∞	700	∞
Train	30900	57260	10	$\perp$	∞	30	∞	∞	∞	∞

Table 5: Comparison of the MAP, ALV, DMC, and HyTech verification systems. Times are expressed in milliseconds. The precision is 10 milliseconds. (i) '0' means termination in less than 10 milliseconds. (ii) '$\perp$' means termination with the answer: 'Unable to verify'. (iii) '∞' means 'No answer' within 100 seconds. (iv) '$\times$' means that the test has not been performed (HyTech has no built-in for checking liveness). For the MAP system we show the total verification times with the *WM* and *WS* generalization operators (see the last two columns of Table 3). For the ALV system we show the times for four options: *default*, *A* (with approximate backward fixpoint computation), *F* (with approximate forward fixpoint computation), and *L* (with computation of loop closures for accelerating reachability). For the DMC system we show the times for two options: *noAbs* (without abstraction) and *Abs* (with abstraction). For the HyTech system we show the times for two options: *Fw* (forward reachability) and *Bw* (backward reachability).

when to apply generalization, and generalization operators, for deciding *how* to generalize. The notions of firing relation and generalization operator extend to CLP the notions of *whistle* algorithm and *most specific generalization* operator, respectively, which have been proposed for positive supercompilation [48] and partial deduction [37]. For defining firing relations we have extended well-binary relations already considered in the program specialization literature, such as the homeomorphic embedding relation [33, 34, 37, 48], and for defining generalization operators we have adapted notions from the area of static program analysis, such as the ones of widening and convex hull [4, 11]. We have also introduced some new notions based on maximal coefficients and sums of coefficients of polynomials.

We have applied our verification method to several examples of infinite state systems taken from the literature, and we have compared the results in terms of precision and efficiency (that is, the number of properties which have been verified and the time taken for verification). On the basis of our experimental results we have found that some generalization strategies outperform the others. In particular, the strategies based on maximal coefficients and sums of coefficients appear to have the best balance between precision and efficiency.

Then, we have applied other tools for the verification of infinite state systems (in particular, ALV [49], DMC [16], and HyTech [29]) to the same set of examples. The experiments show that our specialization-based verification system is quite competitive, especially in terms of precision.

Our approach is closely related to other verification methods for infinite state systems based on the specialization of (constraint) logic programs [36, 38, 41]. However, unlike the approach proposed in [36, 38] we use constraints, which give us very powerful means for dealing with infinite sets of states. The specialization-based verification method presented in [41] consists of one phase only, incorporating top-down query directed specialization and bottom-up answer propagation. That method is restricted to definite constraint logic programs and makes use of a generalization technique which combines widening and convex hull computations for ensuring termination. However, in [41] only two examples of verification have been presented (the Bakery protocol and a Petri net) and no verification times are reported and, thus, it is hard to make an experimental comparison of that method with our method.

Another approach based on program transformation for verifying parameterized (and, hence, infinite state) systems has been presented in [45]. It is an approach based on unfold/fold transformations which are more general than the ones used by us. However, the strategy for guiding the unfold/fold rules proposed in [45] works in fully automatic mode in a small set of examples only.

Finally, we would like to mention that our verification method can be regarded as complementary with respect to the methods for the verification of infinite state systems based on abstraction [2, 4, 9, 12, 16, 26, 27]. These methods work by constructing approximations of the set of reachable states that satisfy a given property. In contrast, the specialization technique applied during Phase (1) of our method, transforms the program for computing sets of states, but it does not change the set of states satisfying the property of interest. Moreover, during Phase (2) we perform an exact computation of the perfect model of the transformed program.

Further enhancements of infinite state verification could be achieved by combining program specialization and abstraction. In particular, an extension of our method could be done by replacing the bottom-up, exact computation of the perfect model performed in Phase (2), by an approximated computation in the style of [4, 16]. However, this extension would require the computation of both over-approximations and under-approximations of models, because of the presence of negation. An interesting direction for future research is the study of how to combine

in the best way, both in terms of precision and efficiency, the verification techniques based on program specialization and the ones based on abstraction.

Acknowledgements

We thank the anonymous referees of CILC 2010 and LOPSTR 2010 for very valuable comments on earlier versions of this paper. We also thank John Gallagher for providing the code of some of the systems considered in Section 5.

This work has been partially supported by ERCIM and PRIN.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *Proceedings of the IEEE Symposium on Logic in Computer Science, LICS'96*, pp. 313–321, IEEE Computer Society Press, 1996.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, "Monotonic abstraction (On efficient verification of parameterized systems)," *International Journal of Foundations of Computer Science*, vol. 20, no. 5, pp. 779–801, 2009.

[3] K. R. Apt and R. N. Bol, "Logic programming and negation: A survey," *Journal of Logic Programming*, vol. 19, 20, pp. 9–71, 1994.

[4] G. Banda and J. P. Gallagher, "Constraint-based abstraction of a model checker for infinite state systems," in *23rd Workshop on Constraint Logic Programming (WLP 2009)*, Institute of Computer Science, Potsdam University, 2009.

[5] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci, "FAST: Acceleration from theory to practice," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 5, pp. 401–424, 2008.

[6] T. Bultan, "Bdd vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems," in *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2000), Berlin, Germany, March 25 - April 2, 2000*, Lecture Notes in Computer Science 1785, pp. 441–455, Springer, 2000.

[7] 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*, vol. 21, no. 4, pp. 747–789, 1999.

[8] W. Chen and D. S. Warren, "Tabled evaluation with delaying for general logic programs," *JACM*, vol. 43, no. 1, 1996.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, 1994.

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

[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)*, pp. 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*, vol. 19, no. 2, pp. 253–291, 1997.

[13] G. Delzanno, "Constraint-based verification of parameterized cache coherence protocols.," *Formal Methods in System Design*, vol. 23, no. 3, pp. 257–301, 2003.

[14] G. Delzanno and T. Bultan, "Constraint-based verification of client-server protocols," in *Principles and Practice of Constraint Programming (CP 2001)* (T. Walsh, ed.), Lecture Notes in Computer Science 2239 , pp. 286–301, Springer-Verlag, 2001.

[15] G. Delzanno, J. Esparza, and A. Podelski, "Constraint-based analysis of broadcast protocols," in *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL '99)* (J. Flum and M. Rodríguez-Artalejo, eds.), Lecture Notes in Computer Science 1683, (London, UK), pp. 50–66, Springer-Verlag, 1999.

[16] G. Delzanno and A. Podelski, "Constraint-based deductive model checking.," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 3, pp. 250–270, 2001.

[17] J. Esparza, "Decidability of model checking for infinite-state concurrent systems," *Acta Informatica*, vol. 34, no. 2, pp. 85–107, 1997.

[18] S. Etalle and M. Gabbrielli, "Transformations of CLP modules," *Theoretical Computer Science*, vol. 166, pp. 101–146, 1996.

[19] 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, pp. 85–96, University of Southampton, UK, 2001.

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

[21] F. Fioravanti, A. Pettorossi, and M. Proietti, "Verifying infinite state systems by specializing constraint logic programs," R. 657, IASI-CNR, Roma, Italy, 2007.

[22] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni, "Program specialization for verifying infinite state systems: An experimental evaluation," in *Preliminary Proceedings of the 20th International Symposium on Logic-Based Synthesis and Transformation (LOPSTR 2010), July 23-25, 2010, Hagenberg, Austria* (M. Alpuente, ed.), RISC-Linz Report Series No. 10-14, pp. 93–112, 2010.

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

[24] L. Fribourg and H. Olsén, "Proving safety properties of infinite state systems by compilation into Presburger arithmetic," in *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR '97)*, Lecture Notes in Computer Science 1243, pp. 96–107, Springer-Verlag, 1997.

[25] J. P. Gallagher, "Tutorial on specialisation of logic programs," in *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pp. 88–98, ACM Press, 1993.

[26] G. Geeraerts, J.-F. Raskin, and L. Van Begin, "Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS," *J. Comput. Syst. Sci.*, vol. 72, no. 1, pp. 180–203, 2006.

[27] 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, pp. 426–440, Springer, 2001.

[28] N. Halbwachs, Y.-E. Proy, and P. Roumanoff, "Verification of real-time systems using linear relation analysis," in *Formal Methods in System Design*, pp. 157–185, 1997.

[29] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997.

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

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

[32] D. Lesens and H. Saïdi, "Abstraction of parameterized networks," *Electronic Notes of Theoretical Computer Science*, vol. 9, p. 41, 1997.

[33] M. Leuschel, "Improving homeomorphic embedding for online termination," in *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '98), Manchester, UK, June 1998* (P. Flener, ed.), Lecture Notes in Computer Science 1559, pp. 199–218, Springer-Verlag, 1999.

[34] M. Leuschel, "Homeomorphic embedding for online termination of symbolic methods," in *The Essence of Computation*, Lecture Notes in Computer Science 2566, pp. 379–403, Springer, 2002.

[35] M. Leuschel and M. Bruynooghe, "Logic program specialisation through partial deduction: Control issues," *Theory and Practice of Logic Programming*, vol. 2, no. 4&5, pp. 461–515, 2002.

[36] M. Leuschel and H. Lehmann, "Coverability of reset Petri nets and other well-structured transition systems by partial deduction.," in *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July* (J. W. Lloyd, ed.), Lecture Notes in Artificial Intelligence 1861, pp. 101–115, Springer-Verlag, 2000.

[37] M. Leuschel, B. Martens, and D. De Schreye, "Controlling generalization and polyvariance in partial deduction of normal logic programs," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 1, pp. 208–258, 1998.

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

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

[40] U. Nilsson and J. Lübcke, "Constraint logic programming for local and symbolic model-checking," in *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July* (J. W. Lloyd, ed.), Lecture Notes in Artificial Intelligence 1861, pp. 384–398, Springer-Verlag, 2000.

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

[42] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[43] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, Lecture Notes in Computer Science 1102, pp. 184–195, Springer-Verlag, 1996.

[44] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren, "Efficient model checking using tabled resolution," in *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, Lecture Notes in Computer Science 1254, pp. 143–154, Springer-Verlag, 1997.

[45] A. Roychoudhury, K. N. 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, pp. 172–187, Springer, 2000.

[46] H. Seki, "Unfold/fold transformation of stratified programs," *Theoretical Computer Science*, vol. 86, pp. 107–139, 1991.

[47] H. B. Sipma, T. E. Uribe, and Z. Manna, "Deductive model checking," *Formal Methods in System Design*, vol. 15, pp. 49–74, 1999.

[48] M. H. Sørensen and R. Glück, "An algorithm of generalization in positive supercompilation," in *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)* (J. W. Lloyd, ed.), pp. 465–479, MIT Press, 1995.

[49] T. Yavuz-Kahveci and T. Bultan, "Action Language Verifier: An infinite-state model checker for reactive software specifications," *Formal Methods in System Design*, vol. 35, no. 3, pp. 325–367, 2009.