ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

F. Fioravanti, A. Pettorossi, M. Proietti

VERIFYING INFINITE STATE SYSTEMS BY SPECIALIZING CONSTRAINT LOGIC PROGRAMS

R. 657 Febbraio 2007

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 : `proietti@iasi.rm.cnr.it`.
URL : `http://www.iasi.rm.cnr.it/~proietti`.

ISSN: 1128–3378

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

viale Manzoni 30, 00185 ROMA, Italy

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

Abstract

We propose a method for the specification and the automated verification of temporal properties of infinite state reactive systems. Given a reactive system $\mathcal{K}$ and a formula φ of the branching time temporal logic CTL, we construct a locally stratified constraint logic program $P_{\mathcal{K}}[\varphi]$ such that the system $\mathcal{K}$ verifies φ if and only if $prop \in M(P_{\mathcal{K}}[\varphi])$, where *prop* is a predicate symbol defined in $P_{\mathcal{K}}[\varphi]$ and $M(P_{\mathcal{K}}[\varphi])$ is the perfect model of $P_{\mathcal{K}}[\varphi]$. Then we check whether or not $prop \in M(P_{\mathcal{K}}[\varphi])$ by specializing the program $P_{\mathcal{K}}[\varphi]$ w.r.t. *prop* and deriving a new program P_{sp} containing *either* the fact $prop \leftarrow$ (in which case the temporal formula φ is verified by the system) *or* no clause for *prop* (in which case the temporal formula φ is not verified by the system). Our specialization method makes use of: (i) a set of specialization rules that preserve the perfect model of constraint logic programs, and (ii) an automatic strategy that guides the application of these rules for deriving the specialized program P_{sp}. Our strategy always terminates and is sound for verifying CTL formulas. Due to the undecidability of CTL formulas in the case of infinite state systems, our strategy is incomplete, that is, we may derive a specialized program P_{sp} containing a clause for *prop* different from the fact $prop \leftarrow$. However, as indicated by the results we have obtained by using our prototype verification system, our strategy allows us to verify a large collection of properties of infinite state systems.

Key words: Verification of reactive systems, program specialization, constraint logic programming.

1. Introduction

Model checking is a highly successful technique for the automatic verification of properties of *finite state* reactive systems [12]. In model checking the time evolution of a reactive system is modeled as a binary transition relation on a finite set of states, and the properties of that evolution are expressed as propositional temporal formulas. Model checking is very useful in practice, because the problem of checking whether or not a given propositional temporal formula is true for a finite transition relation, is decidable. In particular, there are very efficient model checking algorithms for the *Computation Tree Logic* (CTL, for short), which is a very expressive branching time propositional temporal logic where one can describe, among others, the so-called *safety* and *liveness* properties of reactive systems [12].

One of the most challenging problems in the area of verification of reactive systems, is the extension of the model checking technique to *infinite state* reactive systems (see, for instance, [43]). In this case, a reactive system is modeled as a binary transition relation over an infinite set of states. Unfortunately, when considering model checking of infinite state systems, the verification problem is undecidable for most classes of temporal formulas.

In recent years three main approaches have been followed for trying to overcome that undecidability limitation.

The first approach consists in considering *decidable subclasses* of systems and formulas (see, for instance, [1, 22, 47]). By following this approach one may provide fully automatic techniques that, however, are not applicable outside the restricted classes of systems and properties considered.

The second approach consists in enhancing finite state model checking with more general *deductive* techniques (see, for instance, [48, 52, 63, 64]). This approach provides a greater generality, but it needs some degree of human guidance which is often difficult to provide when dealing with complex reactive systems.

The third approach consists in designing methods based on *abstractions*, that is, mappings by which one can reduce an infinite state system (or a finite state system with a large number of states) to a finite state system with the same properties of interest and, hopefully, easier to verify (see, for instance, [2, 11, 16, 68]). Once an abstraction is provided, these techniques are fully automatic, but the choice of the suitable abstraction cannot be always mechanized.

We propose a verification method that combines the generality of the approaches based on deduction with the mechanizability of the approaches based on abstractions. By merging and extending the techniques presented in [19, 27, 55], our method makes use of *constraint logic programs* (or *CLP programs*, for short) with negation [6, 32] for specifying infinite state systems and their properties. Our method is automatic, sound, but incomplete, and its main novelty resides in the idea of making use of some *program specialization* techniques [25, 29, 34, 37] for CLP programs as an inference mechanism for checking the properties of interest.

In our method the transition relation that models the system under consideration is specified by a finite collection of constraints over pairs of states. The use of constraints allows us to model an infinite state system in a simple way, because a single constraint can represent the possibly infinite set of pairs of states that satisfy it. Given an infinite state system $\mathcal{K}$ specified by constraints, the temporal properties of $\mathcal{K}$ are encoded as a CLP program $P_{\mathcal{K}}$ which defines a binary predicate $sat(s, \varphi)$, meaning that in the state s the CTL formula φ holds.

In order to encode negated CTL formulas, the program $P_{\mathcal{K}}$ uses *locally stratified* negation, that is, if we consider the set of the ground instances of clauses in $P_{\mathcal{K}}$ no ground atom depends negatively on itself [6]. The semantics of $P_{\mathcal{K}}$ is provided by its *perfect model*, denoted $M(P_{\mathcal{K}})$.

(Recall that for a program with locally stratified negation the perfect model is unique and it is equal to the unique *stable model* and to the *well-founded model* [6].) We will show that $P_{\mathcal{K}}$, with the perfect model semantics, provides an adequate encoding of the temporal properties of $\mathcal{K}$, and thus, we may check whether or not the temporal formula φ holds in state s of system $\mathcal{K}$ by checking whether or not $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$.

Unfortunately, the proof procedures normally used in constraint logic programming, such as SLDNF resolution [6, 41] or tabled resolution [10, 59], often diverge when trying to check whether or not $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$ for an infinite state system $\mathcal{K}$. This is due to the limited ability of these proof procedures to cope with infinitely failed derivations.

In order to overcome the limitations of SLDNF resolution and tabled resolution, in this paper we propose a method for the verification of infinite state systems based on *program specialization*. Program specialization is a program transformation technique whose objective is the adaptation of a program to the context of use (see, for instance, [25, 29, 34, 37]). By applying our program specialization technique we will be able to check whether or not $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$ for many nontrivial formulas φ and systems $\mathcal{K}$ in which neither SLDNF resolution nor tabled resolution are successful.

Now we briefly indicate how program specialization can be used for checking whether or not, for all initial states s of a given system $\mathcal{K}$, the atom $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$. We start off by considering the task of checking the equivalent property that there exists no initial state s such that $\neg\varphi$ holds in s. A predicate *prop* that encodes this property can be defined by the following two clauses:

γ_1 : $prop \leftarrow \neg negprop$
γ_2 : $negprop \leftarrow sat(X, init \wedge \neg\varphi)$

where: (i) *init* is a property that holds in a state X iff X is an initial state of the system $\mathcal{K}$, and (ii) *negprop* is a new predicate symbol. We have that:

$$\text{for all initial states } s,\ sat(s, \varphi) \in M(P_{\mathcal{K}}) \quad \text{iff} \quad prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$$

In order to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ we first transform the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ into an equivalent program P_{sp} such that,

$$\text{for all initial states } s,\ sat(s, \varphi) \in M(P_{\mathcal{K}}) \quad \text{iff} \quad prop \in M(P_{sp}) \qquad (\dagger)$$

The transformation of $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ into P_{sp} is performed by specializing $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ w.r.t. *prop*.

At the end of the specialization process, we check whether or not the specialized program P_{sp} satisfies one of the following two properties:
(T) P_{sp} contains the clause $prop \leftarrow$
(F) P_{sp} does *not* contain any clause for the predicate *prop*.
If Property (T) is satisfied, then we infer that property *prop* holds, that is, for all initial states s of a system $\mathcal{K}$, $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$. If Property (F) is satisfied, then we infer that property *prop* does *not* hold, that is, there exists an initial state s such that $sat(s, \neg\varphi)$ belongs to $M(P_{\mathcal{K}})$.

Our specialization technique follows the approach based on transformation rules and strategies advocated in [9]. The transformation rules we will present below, are variants of the familiar unfolding, folding, clause deletion, and constraint replacement rules presented in the literature. Various versions of these sets of rules can be found in [8, 23, 26, 42, 61, 66]. The rules we will use in this paper preserve the perfect model semantics [26], and thus, they ensure that the above equivalence (†) holds. This fact validates our method for verifying whether or not a property holds in a given system. We will also present a transformation strategy that guides the

application of the transformation rules with the aim of deriving a program P_{sp} satisfying either Property (T) or Property (F). Our strategy is fully automatic and always terminates, and thus, due to the above mentioned undecidability limitations, it is incomplete. Indeed, it may be the case that our strategy derives a program P_{sp} which satisfies neither Property (T) nor Property (F), that is, P_{sp} contains some clauses for *prop* which are different from *prop* $\leftarrow$. In this case we are not able to tell whether or not $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$.

In order to ensure termination, our strategy uses a *generalization* technique that is related to some *abstract interpretation* techniques [13] and it plays a role similar to that of abstraction in the verification methods described in [11, 16, 2]. However, since it is applied *during* the verification process, and *not before* the verification process, our generalization is often more flexible than abstraction.

The main technical contributions of this paper are the following ones. (i) We have shown that the CTL properties of the *concurrent systems* defined in [62], can be expressed by using locally stratified CLP programs. (ii) We have defined variants of the usual transformation rules (such as unfolding, folding, clause deletion, and constraint replacement) which are suitable for performing the specialization of locally stratified CLP programs, and we have shown that these rules preserve the perfect model semantics. (iii) We have proposed an automatic strategy for program specialization and, in particular, a technique for generalization that makes program specialization always terminating. (iv) Finally, we have demonstrated that our technique is powerful enough to automatically verify safety and liveness properties of many infinite state systems considered in the literature.

The structure of the paper is as follows. In Section 2 we recall some preliminary notions concerning locally stratified constraint logic programs and the CTL temporal logic. In Section 3 we consider a class of reactive systems and we show how the CTL properties of the systems in that class can be encoded by using locally stratified CLP programs. In Section 4 we present the transformation rules we use for specializing programs and we prove the correctness of these rules w.r.t. the perfect model semantics. In Section 5 we describe our strategy for program specialization, and we describe the generalization technique we use for ensuring the termination of the strategy. In Section 6 we report on some experiments of automatic protocol verification we have done by using a prototype implementation of our method on the MAP transformation system [44]. In particular, we have proved safety and liveness properties of mutual exclusion protocols, parameterized cache coherence protocols, and reset Petri nets. In Section 7, we show how to generate witnesses of Finally, in Section 8 we compare our work with other verification techniques proposed in the literature. Among them we devote special attention to those techniques that use logic programming, constraints, tabled resolution, program analysis, and program transformation [19, 24, 27, 28, 38, 40, 49, 55, 57, 58].

2. Preliminaries

In this section we recall some basic notions of constraint logic programming. For notions not defined here the reader may refer to [4, 6, 32, 33, 41]. We also present our notational conventions and we briefly recall the syntax and the semantics of the Computation Tree Logic (CTL, for short), which is the logic we use for expressing properties of reactive systems. For a more detailed treatment of CTL the reader may look at [12].

2.1. Syntax of Constraint Logic Programs

We consider a first order language $\mathcal{L}$ generated by an infinite set *Vars* of *variables*, a set *Funct* of *function symbols* with arity, and a set *Pred* of *predicate symbols* with arity. We assume that *Pred* is the union of two disjoint sets: (i) the set $Pred_c$ of *constraint* predicate symbols, including *true*, *false*, and the *equality* symbol $=$, and (ii) the set $Pred_u$ of *user defined* predicate symbols.

A *term* of $\mathcal{L}$ is either a variable or an expression of the form $f(t_1, \ldots, t_n)$, where f is a symbol of arity $n \geq 0$ in *Funct* and $t_1, \ldots, t_n$ are terms. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a symbol of arity $n \geq 0$ in *Pred* and $t_1, \ldots, t_n$ are terms. A *formula* of $\mathcal{L}$ is either an atom or a formula constructed, as usual, from already constructed formulas by means of connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$) and quantifiers ($\exists$, $\forall$).

Let e be a term or a formula. The set of variables occurring in e is denoted by $vars(e)$. Similar notation will be used for denoting the set of variables occurring in a set of terms or formulas. Given a formula φ, the set of the *free variables* in φ is denoted by $FV(\varphi)$. A term or a formula is *ground* if it contains no variables. Given a set $X = \{X_1, \ldots, X_n\}$ of variables, by $\forall X\, \varphi$ we denote the formula $\forall X_1 \ldots \forall X_n\, \varphi$. By $\forall(\varphi)$ we denote the *universal closure* of φ, that is, the formula $\forall X\, \varphi$, where $FV(\varphi) = X$. Analogously, by $\exists(\varphi)$ we denote the *existential closure* of φ. We denote a formula φ whose free variables are among $X_1, \ldots, X_n$ also by $\varphi(X_1, \ldots, X_n)$.

An *atomic constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol in $Pred_c$. A *constraint* is either an atomic constraint, or a formula of the form $c_1 \wedge c_2$, where c_1 and c_2 are constraints, or a formula of the form $\exists X\ c$, where c is a constraint. We denote by $\mathcal{C}$ the set of constraints of the language $\mathcal{L}$.

An *atom* is an atomic formula $p(t_1, \ldots, t_n)$ where p is an element of $Pred_u$ and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom A, also called *positive literal*, or a negated atom $\neg\, A$, also called *negative literal*. A *goal* is a (possibly empty) conjunction of literals (here we depart from [41], where a goal is defined as the negation of a conjunction of literals). A *constrained literal* is the conjunction of a constraint and a literal. A *constrained goal* is the conjunction of a constraint and a goal. The empty conjunction of constraints or literals is identified with *true*.

A *clause* γ is a formula of the form $H \leftarrow c \wedge G$, where: (i) H is an atom, called the *head* of γ and denoted $hd(\gamma)$, and (ii) $c \wedge G$ is a constrained goal, called the *body* of γ and denoted $bd(\gamma)$. If H is of the form $p(t_1, \ldots, t_n)$ we say that clause γ is a *clause for* p. Clauses of the form $H \leftarrow c$, where c is a constraint, are called *constrained facts*. Clauses of the form $H \leftarrow true$ are called *facts*, and they are also written as $H \leftarrow$.

A *constraint logic program* (or *program*, for short) is a finite set of clauses. A *definite* constraint logic program is a finite set of clauses whose bodies have no occurrences of negative literals.

Given two atoms A and B, we denote by $A = B$ the following constraint: (i) $t_1 = u_1 \wedge \ldots \wedge t_n = u_n$, 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. We say that a term t is free for a variable X in a formula φ if by substituting all free occurrences of X in φ by t, we do not introduce new occurrences of bound variables. A formula ψ is an *instance* of a formula φ if ψ is obtained from φ by applying a substitution $\{X_1/t_1, \ldots, X_n/t_n\}$ such that, for $i = 1, \ldots, n$, the term t_i is free for X_i in φ.

We say that a predicate p *immediately depends on* a predicate q in a program P, and we write $p\ dep_P\ q$, if in P there exists a clause of the form $p(\ldots) \leftarrow B$ such that an atom of the form $q(\ldots)$ occurs in B. Let dep_P^+ be the transitive closure of the dep_P relation. We say that p *depends on* q in P if $p\ dep_P^+\ q$.

Given a user defined predicate symbol p and a program P, the *definition of* p *in* P, denoted $Def(p, P)$, is the set of clauses γ in P such that p is the predicate symbol of $hd(\gamma)$. By $Def^*(p, P)$

we denote the set $Def(p, P) \cup \{\gamma \in P \mid p \text{ depends on the predicate symbol of } hd(\gamma)\}$.

A *variable renaming* is a bijective mapping from *Vars* to *Vars*. The application of a variable renaming ρ to a formula φ returns the formula $\rho(\varphi)$, which is said to be a *variant* of φ, obtained by replacing each (bound or free) variable X in φ by the variable $\rho(X)$. A variant of a set $\{\varphi_1, \ldots, \varphi_n\}$ of formulas is a set $\{\varphi'_1, \ldots, \varphi'_n\}$, where, for $i = 1, \ldots, n$, φ'_i is a variant of φ_i.

We will feel free to apply to clauses the following transformations that, as the reader may verify, preserve the perfect model semantics (see Section 2.2):
(1) application of variable renamings,
(2) reordering of constraints and literals in the body (we will usually move all constraints to the left of the body and all literals to the right of the body), and
(3) replacement of a clause of the form $H \leftarrow X = t \wedge c \wedge G$, where $X \notin vars(t)$, by the clause $(H \leftarrow c \wedge G)\{X/t\}$, and vice versa.

2.2. Semantics of Constraint Logic Programs

Now we introduce the notions of local stratification and perfect model for constraint logic programs. These notions are extensions of similar notions for logic programs [6, 54] and they are parametric w.r.t. the interpretation of the constraints [32, 33].

A *constraint interpretation* $\mathcal{D}$ consists of: (1) a non-empty set D, called *carrier*, (2) an assignment of a function $f_{\mathcal{D}}$: $D^n \rightarrow D$ to each n-ary function symbol f in *Funct*, and (3) an assignment of a relation $p_{\mathcal{D}}$ over D^n to each n-ary constraint predicate symbol p in $Pred_c$. We say that f *is interpreted as* $f_{\mathcal{D}}$ and p *is interpreted as* $p_{\mathcal{D}}$. In particular, for any constraint interpretation $\mathcal{D}$, *true* is interpreted as the relation D^0, that is, the singleton $\{\langle\rangle\}$ whose unique element is the empty tuple, *false* is interpreted as the empty set, and the equality symbol $=$ is interpreted as the set $\{\langle d, d\rangle \mid d \in D\}$.

We assume that D is a set of ground terms. This is not restrictive because we may enlarge the language $\mathcal{L}$ by making every element of D to be an element of the set *Funct* of function symbols. Sometimes in the sequel, by abuse of language, we will identify the constraint interpretation $\mathcal{D}$ with its carrier D.

Given a formula φ where all predicate symbols belong to $Pred_c$, we consider the satisfaction relation $\mathcal{D} \models \varphi$, which is defined as usual in the first order predicate calculus (see, for instance, [4]).

An interpretation of the predicate symbols in $Pred_u$ is called a $\mathcal{D}$-*interpretation* and is defined as follows. Given a constraint interpretation $\mathcal{D}$, a $\mathcal{D}$-interpretation I assigns a relation over D^n to each n-ary predicate symbol in $Pred_u$, that is, I is a subset of the set $\mathcal{B}_{\mathcal{D}}$ defined as follows:

$$\mathcal{B}_{\mathcal{D}} = \{p(d_1, \ldots, d_n) \mid p \text{ is a predicate symbol in } Pred_u \text{ and } \langle d_1, \ldots, d_n\rangle \in D^n\}$$

Now we define the notion of $\mathcal{D}$-*model* of a program, which is a $\mathcal{D}$-interpretation where all clauses of the program are true. First we need to introduce the notion of a *valuation*, that is, a function v: *Vars* $\rightarrow D$. Then we extend the domain of v to terms, constraints, literals, and clauses as we now indicate. Given a term t, we inductively define the term $v(t)$ as follows: (i) if t is a variable X then $v(t) = v(X)$, and (ii) if t is $f(t_1, \ldots, t_n)$ then $v(t) = f_{\mathcal{D}}(v(t_1), \ldots, v(t_n))$. Given a constraint c, $v(c)$ is the constraint obtained by replacing every free variable $X \in FV(c)$ by the ground term $v(X)$. Note that $v(c)$ is a closed (possibly not ground) formula. Given a literal L, (i) if L is the atom $p(t_1, \ldots, t_n)$, then $v(L)$ is the ground atom $p(v(t_1), \ldots, v(t_n))$, and (ii) if L is the negated atom $\neg A$, then $v(L)$ is the ground literal $\neg v(A)$. Given a clause γ: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, $v(\gamma)$ is the clause $v(H) \leftarrow v(c) \wedge v(L_1) \wedge \ldots \wedge v(L_m)$.

Let I be a $\mathcal{D}$-interpretation and v a valuation. A literal $v(L)$ *is true in* I if either (i) L is an atom and $v(L) \in I$, or (ii) L is a negated atom $\neg A$ and $v(A) \notin I$. A literal $v(L)$ *is false in* I if it is not true in I. Given a clause γ: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, $v(\gamma)$ *is true in* I if either (i) $v(H)$ is true in I, or (ii) $\mathcal{D} \not\models v(c)$, or (iii) there exists $i \in \{1, \ldots, m\}$ such that $v(L_i)$ is false in I.

A $\mathcal{D}$-interpretation M is a $\mathcal{D}$-*model* of a program P if for every clause γ in P and for every valuation v, we have that $v(\gamma)$ is true in M. It can be shown that every definite constraint logic program P has a *least* $\mathcal{D}$-model w.r.t. set inclusion (see, for instance, [33]). This least $\mathcal{D}$-model can be constructed as the least fixpoint of the *immediate consequence* function T_P from the set of $\mathcal{D}$-interpretations to the set of $\mathcal{D}$-interpretations, defined as follows:

$$\begin{aligned} T_P(I) = \{H \mid & \text{ there exist a valuation } v \text{ and a clause } \gamma \text{ in } P \text{ such that:} \\ & \text{(i) } v(\gamma) = (H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m), \text{ (ii) } \mathcal{D} \models c, \text{ and} \\ & \text{(iii) } L_1, \ldots, L_m \text{ are true in } I\} \end{aligned}$$

It can be shown that T_P is a monotonic function over the complete lattice of $\mathcal{D}$-interpretations, ordered by set inclusion, and its least fixpoint, denoted by $lfp(T_P)$, is the least $\mathcal{D}$-model of P [33]. Moreover, since T_P is continuous, it has a least fixpoint which can be constructed by iterating the application of T_P starting from the empty $\mathcal{D}$-interpretation, that is:

$$lfp(T_P) = T_P^{\omega}(\emptyset)$$

where ω is the first limit ordinal [4, 33, 41].

Unfortunately, for some constraint logic program P which are not definite, the function T_P is not monotonic and P does not have a least $\mathcal{D}$-model. For example, the program consisting of the clause $p \leftarrow \neg q$ has the two non-least $\mathcal{D}$-models $\{p\}$ and $\{q\}$. A well-known approach followed in logic programming to overcome this difficulty consists in considering particular classes of programs with suitable restrictions such that we are still able to associate a unique model to every program in any of these classes. Here we consider the classes of: (i) *stratified* programs, and (ii) *locally stratified* programs, and now we show how to construct their *perfect models* [6, 54]. We need the following definitions.

A *level mapping* is a function from the set of predicate symbols to the set of finite ordinals. Given a level mapping λ, we extend it to literals as follows: if L is an atom $p(\ldots)$ or a negated atom $\neg p(\ldots)$, then $\lambda(L) = \lambda(p)$. A clause γ of the form $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$ is *stratified* w.r.t. a level mapping λ if for $i = 1, \ldots, m$, if L_i is a positive literal then $\lambda(H) \geq \lambda(L_i)$, otherwise, if L_i is a negative literal then $\lambda(H) > \lambda(L_i)$. A program P is *stratified* if there exists a level mapping λ such that every clause of P is stratified w.r.t. λ. If a program P is stratified w.r.t. λ, then there exists a finite sequence $S_1, \ldots, S_k$ of sets of clauses, called a *stratification* of P, such that: (i) $P = S_1 \cup \ldots \cup S_k$, and (ii) for any two clauses α and β in P, for any two clauses α and β in P, $\lambda(hd(\alpha)) < \lambda(hd(\beta))$ iff there exist i and j such that: (a) $1 \leq i < j \leq k$, (b) $\alpha \in S_i$, and (c) $\beta \in S_j$. The sets $S_1, \ldots, S_k$ are said to be the *strata* of P. Note that, as a consequence of the definition, the strata of a program are pairwise disjoint sets of clauses and two clauses with the same head predicate belong to the same stratum.

Thus, in a clause γ of a stratified program P the predicate appearing in a negative literal of the body of γ does not depend in P on the predicate of the head of γ. For instance, the program consisting of the clause $p \leftarrow \neg q$ is stratified w.r.t. any level mapping λ such that $\lambda(p) > \lambda(q)$. It has been shown that we can construct a unique *standard model* for any given stratified logic program by working bottom-up on its strata [6]. The construction of the standard model of a stratified program is generalized by the construction of the perfect model of a *locally stratified* program that we now present.

A *local stratification* is a function σ: $\mathcal{B}_\mathcal{D} \to W$, where W is the set of countable ordinals. If $A \in \mathcal{B}_\mathcal{D}$ and $\sigma(A) = \alpha$ we say that the *stratum* of A is α, or A is in stratum α. Given a clause γ in P, a valuation v, and a local stratification σ, we say that the clause $v(\gamma)$ of the form: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$ is *locally stratified* w.r.t. σ if either $\mathcal{D} \not\models c$ or, for $i = 1, \ldots, m$, if L_i is an atom A then $\sigma(H) \geq \sigma(A)$, otherwise, if L_i is a negated atom $\neg A$ then $\sigma(H) > \sigma(A)$. Given a local stratification σ, we say that program P is *locally stratified w.r.t.* σ if for every clause γ in P and for every valuation v, the clause $v(\gamma)$ is locally stratified w.r.t. σ. A program P is *locally stratified* if there exists a local stratification σ such that P is *locally stratified w.r.t.* σ.

Thus, a local stratification of a program P corresponds to a, possibly infinite, stratification of the following set of ground clauses:

$$\{H \leftarrow L_1 \wedge \ldots \wedge L_m \mid \text{there exist a valuation } v \text{ and a clause } \gamma \text{ in } P \text{ such that } v(\gamma) \text{ is } H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m \text{ and } \mathcal{D} \models c\}$$

We have that a stratified program is also locally stratified, but not vice versa. For instance, let us consider the following program *Even*:

$$even(0) \leftarrow$$
$$even(X) \leftarrow X = Y + 1 \wedge \neg even(Y)$$

where the constraint interpretation is as follows: (1) the carrier is the set $\mathbb{N}$ of the natural numbers, (2) the addition function is assigned to the function symbol $+$, and (3) the identity relation is assigned to the equality predicate. The program *Even* is not stratified, but it is locally stratified w.r.t. σ defined as follows: $\sigma(even(n)) = n$, for every $n \in \mathbb{N}$.

Now we present the definition of the perfect model of a locally stratified program. Our presentation is slightly different from those in [6, 54] for two reasons: (i) we consider constraint logic programs, instead of logic programs, and (ii) we do not rely on the *preference relation* between Herbrand interpretations. However, it can be shown that, for every logic program without constraints, our construction generates a model which is equal to the one defined in [6, 54].

The perfect model $M(P)$ of a locally stratified constraint logic program P is constructed by generalizing the immediate consequence function T_P to a function $T_{P,\alpha}$ that depends also on a stratum α. The function $T_{P,\alpha}$ is defined as follows. Given a constraint logic program P which is locally stratified w.r.t. a local stratification σ, and an interpretation I, we have that:

$$T_{P,\alpha}(I) = \{H \mid \text{there exist a valuation } v \text{ and a clause } \gamma \text{ in } P \text{ such that: (i) } v(\gamma) \text{ is } H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m, \text{ (ii) } \sigma(H) = \alpha, \text{ (iii) } \mathcal{D} \models c, \text{ and (iv) for } i = 1, \ldots, m, \textit{ if } \sigma(L_i) = \alpha \textit{ then } L_i \text{ is true in } I \textit{ else } L_i \text{ is true in } \textit{lfp}(T_{P,\sigma(L_i)})\}$$

Informally, $T_{P,\alpha}(I)$ computes the set of immediate consequences of I which are in stratum α, knowing the truth value of the literals in stratum β, for $\beta < \alpha$. Indeed, at Condition (iv) of the above definition of $T_{P,\alpha}$, we have that if $\lambda \neq \alpha$ then $\lambda < \alpha$, because the program P is locally stratified.

Note that in this definition of $T_{P,\alpha}$ we refer to the least fixpoint of $T_{P,\beta}$, for all $\beta < \alpha$. The following result ensures that the least fixpoint of $T_{P,\beta}$ is defined for all ordinals β, and thus, $T_{P,\alpha}$ is a total function on the lattice of $\mathcal{D}$-interpretations.

Theorem 2.1. *Given a locally stratified program P, for every countable ordinal $\alpha \in W$, $T_{P,\alpha}$ is a monotonic function on the complete lattice of $\mathcal{D}$-interpretations and, thus, $T_{P,\alpha}$ has a least fixpoint $\textit{lfp}(T_{P,\alpha})$. Moreover, $T_{P,\alpha}$ is continuous and, thus, $\textit{lfp}(T_{P,\alpha}) = T^\omega_{P,\alpha}(\emptyset)$.*

Proof: The proof proceeds by well-founded induction on the ordinal α. Assume that, for all $\beta < \alpha$, $T_{P,\beta}$ is a monotonic function on the complete lattice of $\mathcal{D}$-interpretations. Then, by the Knaster-Tarski Theorem (see, for instance, [4, 41]) $T_{P,\beta}$ has a least fixpoint $lfp(T_{P,\beta})$. Hence, $T_{P,\alpha}$ is a total function. Let us now show that $T_{P,\alpha}$ is monotonic. Let I, J be $\mathcal{D}$-interpretations such that $I \subseteq J$. Let us consider any valuation v and any clause γ in P. Let $v(\gamma)$ be of the form: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$. For $i = 1, \ldots, m$, if L_i is true in I then L_i is true in J, and thus, if $H \in T_{P,\alpha}(I)$ then $H \in T_{P,\alpha}(J)$.

Finally, we show that, for every countable ordinal $\alpha \in W$, $T_{P,\alpha}$ is continuous, by proving that, for every ω-chain $I_0 \subseteq I_1 \subseteq \ldots$ of $\mathcal{D}$-interpretations [4],

$$T_{P,\alpha}(\bigcup_{n<\omega} I_n) \subseteq \bigcup_{n<\omega} T_{P,\alpha}(I_n)$$

Suppose that $A \in T_{P,\alpha}(\bigcup_{n<\omega} I_n)$. Then, by the definition of $T_{P,\alpha}$, there exists a valuation v and a clause γ in P such that: (i) $v(\gamma) = (A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m)$, (ii) $\sigma(A) = \alpha$, (iii) $\mathcal{D} \models c$, and (iv) for $i = 1, \ldots, m$, if $\sigma(L_i) = \alpha$ then L_i is true in $\bigcup_{n<\omega} I_n$ else if $\sigma(L_i) = \beta < \alpha$ then L_i is true in $lfp(T_{P,\beta})$. L_i is true in $\bigcup_{n<\omega} I_n$ iff there exists $n < \omega$ such that L_i is true in I_n. Hence, there exists $\overline{n}$ such that, for $i = 1, \ldots, m$, if $\sigma(L_i) = \alpha$ then L_i is true in $I_{\overline{n}}$, and if $\sigma(L_i) = \beta < \alpha$ then L_i is true in $lfp(T_{P,\beta})$. Thus, $A \in T_{P,\alpha}(I_{\overline{n}})$, and therefore, $A \in \bigcup_{n<\omega} T_{P,\alpha}(I_n)$. □

Now we define the *perfect model* $M(P)$ of a locally stratified constraint logic program P as follows:

$$M(P) = \bigcup_{\alpha \in W} lfp(T_{P,\alpha})$$

The following property is a straightforward consequence of the definitions of $M(P)$ and $T_{P,\alpha}$.

Lemma 2.2. *Let P be a program which is locally stratified w.r.t. a local stratification function σ. Then, for every $A \in \mathcal{B}_{\mathcal{D}}$, $A \in M(P)$ iff $A \in lfp(T_{P,\alpha})$, where $\alpha = \sigma(A)$.*

It can be shown that $M(P)$ is independent of the particular local stratification function we use in the definition of $T_{P,\alpha}$.

In this paper we do not specify any particular algorithm for solving constraints in $\mathcal{C}$. We only assume that there exists a computable total function *solve*: $\mathcal{C} \times \mathcal{P}_{fin}(Vars) \rightarrow \mathcal{C}$, where $\mathcal{P}_{fin}(Vars)$ is the set of all finite subsets of *Vars*. *solve* is assumed to be *sound* w.r.t. constraint equivalence, that is, for every constraint c_1 and for every finite set X of variables, if $solve(c_1, X) = c_2$ then $\mathcal{D} \models \forall X((\exists Y\, c_1) \leftrightarrow c_2)$, where $Y = FV(c_1) - X$ and $FV(c_2) \subseteq FV(\exists Y\, c_1)$. We also assume that *solve* is *complete* w.r.t. satisfiability in the sense that, for any constraint c,

(i) $solve(c, \emptyset) = true$ iff c is *satisfiable*, that is, $\mathcal{D} \models \exists(c)$, and

(ii) $solve(c, \emptyset) = false$ iff c is *unsatisfiable*, that is, $\mathcal{D} \models \neg\exists(c)$.

The soundness and the totality of the *solve* function guarantee the correctness and the termination of the specialization strategy we will present in Section 5. The assumption that the *solve* function is complete w.r.t. satisfiability guarantees that constraint satisfiability is decidable and indeed, constraint satisfiability tests can be performed by applying the *solve* function.

We assume that, for any constraints c_1 and c_2, the entailment relation $\mathcal{D} \models \forall(c_1 \rightarrow c_2)$ is decidable. (This decidability assumption is *not* a consequence of the existence of the *solve* function because, in general, $c_1 \rightarrow c_2$ is not a constraint.)

Note that the definition of the *solve* function considered in this paper includes three different functions described in the literature [32, 33, 45]: (i) a *projection* function that, given a constraint

c and a set W of variables, returns a constraint which is equivalent to $\exists W\, c$ with fewest existential quantifiers, (ii) a *simplification* function which given a constraint, returns its simplified form (such as the solved form of a system of equations), and (iii) a function for testing the *satisfiability* of constraints.

2.3. The Computation Tree Logic

The Computation Tree Logic (CTL, for short) is a propositional temporal logic for expressing properties of the time behavior of a reactive system. This behavior is represented as the tree of the states that the system can reach, and each path of this tree is called a *computation path*. CTL formulas are built from a given set *Elem* of *elementary properties* by using: (i) the connectives: $\neg$ ('not') and $\wedge$ ('and'), (ii) the following linear-time operators along a computation path: G ('always'), F ('sometimes'), X ('nexttime'), and U ('until'), and (iii) the quantifiers over computation paths: A ('for all paths') and E ('for some path'), as indicated by the following definition.

Definition 1 (CTL formulas) *Let Elem be a given set of* elementary properties*. We assume that true and false belong to Elem. A CTL formula φ has the following syntax:*

$$\varphi ::= e \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid EX(\varphi) \mid EU(\varphi_1, \varphi_2) \mid AF(\varphi)$$

where e belongs to Elem.

We also use the following abbreviations:
$EF(\varphi) \equiv EU(true, \varphi)$
$EG(\varphi) \equiv \neg AF(\neg\varphi)$
$AX(\varphi) \equiv \neg EX(\neg\varphi)$
$AU(\varphi_1, \varphi_2) \equiv \neg EU(\neg\varphi_2,\, \neg\varphi_1 \wedge \neg\varphi_2) \wedge \neg EG(\neg\varphi_2)$
$AG(\varphi) \equiv \neg EF(\neg\varphi)$

The semantics of CTL formulas is given by introducing a *Kripke structure* $\mathcal{K}$ and defining the satisfaction relation $\mathcal{K}, s \models \varphi$, which denotes that a formula φ holds in a state s of $\mathcal{K}$. The context will disambiguate between the use of the symbol $\models$ for denoting the satisfaction relation in a Kripke structure and the use of the symbol $\models$ for providing the semantics of constraint logic programs (see Section 2.2).

Definition 2 (Kripke Structure) *A Kripke structure $\mathcal{K}$ is a 4-tuple $\langle S, I, R, L\rangle$ where:*
1. S is a set of states,
2. $I \subseteq S$ is a set of initial states,
3. $R \subseteq S \times S$ is a transition relation*. R is a total relation, that is, for every state $s \in S$ there exists a state $s' \in S$, called a* successor state *of s, such that $s\, R\, s'$, and*
4. $L : S \to \mathcal{P}(Elem)$ is a function that assigns to each state $s \in S$ a subset $L(s)$ of Elem, that is, a set of elementary properties that hold in s. In particular, true, false, and init are elementary properties such that, for all $s \in S$: (i) $true \in L(s)$, (ii) $false \notin L(s)$, and (iii) $init \in L(s)$ iff $s \in I$.
A computation path *in $\mathcal{K}$ starting from a state s_0 is an* infinite *sequence of states $s_0\, s_1 \ldots$ such that $s_i\, R\, s_{i+1}$ for every $i \geq 0$.*

Definition 3 (Satisfaction Relation for a Kripke Structure) *Given a Kripke structure* $\mathcal{K} = \langle S, I, R, L\rangle$*, a state* $s \in S$*, and a formula* φ*, we inductively define the relation* $\mathcal{K}, s \models \varphi$*, denoting that* φ *holds in the state* s *of* $\mathcal{K}$*, as follows:*
$\mathcal{K}, s \models e$ *iff* e *is an elementary property belonging to* $L(s)$
$\mathcal{K}, s \models \neg\varphi$ *iff it is not the case that* $\mathcal{K}, s \models \varphi$
$\mathcal{K}, s \models \varphi_1 \wedge \varphi_2$ *iff* $\mathcal{K}, s \models \varphi_1$ *and* $\mathcal{K}, s \models \varphi_2$
$\mathcal{K}, s \models EX(\varphi)$ *iff there exists a computation path* $s_0\, s_1 \ldots$ *in* $\mathcal{K}$ *such that* $s = s_0$ *and* $\mathcal{K}, s_1 \models \varphi$
$\mathcal{K}, s \models EU(\varphi_1, \varphi_2)$ *iff there exists a computation path* $s_0\, s_1 \ldots$ *in* $\mathcal{K}$ *such that (i)* $s = s_0$ *and (ii) for some* $n \geq 0$ *we have that:* $\mathcal{K}, s_n \models \varphi_2$ *and* $\mathcal{K}, s_j \models \varphi_1$ *for all* $j \in \{0, \ldots, n-1\}$
$\mathcal{K}, s \models AF(\varphi)$ *iff for all computation paths* $s_0\, s_1 \ldots$ *in* $\mathcal{K}$*, if* $s = s_0$ *then there exists* $n \geq 0$ *such that* $\mathcal{K}, s_n \models \varphi$.

3. Expressing CTL Properties by Locally Stratified CLP Programs

In this section we present: (i) a method based on constraints for specifying reactive systems by particular Kripke structures, and (ii) an algorithm for encoding the set of temporal properties of a system as a locally stratified constraint logic program.

Our specification method of Point (i) is basically a constraint-based extension of the method for specifying *concurrent systems* presented in [62]. According to our method, a reactive system is specified by a *constraint-based* Kripke structure as we now define. First, we need the following definition of an *event*.

Definition 4 (Event) *Let* $\mathcal{D}$ *be a constraint interpretation. (i) An* event $t(X, Y)$ *is a constraint of the form* $cond(X) \wedge act(X, Y)$ *such that:*
1. $\mathcal{D} \models \forall X\, (cond(X) \rightarrow \exists Y\, act(X, Y))$*, and*
2. $\mathcal{D} \models \forall X\, \forall Y\, \forall Z\, (act(X, Y) \wedge act(X, Z) \rightarrow Y = Z)$.
The constraint $cond(X)$ *is called the* enabling condition *of the event, and* $act(X, Y)$ *is called the* action *of the event.*
(ii) A disjunction $t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ *of events is* exhaustive *if*

$$\mathcal{D} \models \forall X\, (cond_1(X) \vee \ldots \vee cond_k(X))$$

where, for $i = 1, \ldots, k$*,* $cond_i(X)$ *is the enabling condition of* $t_i(X, Y)$.

Conditions 1 and 2 mean that the relation *act* is a function of its first argument X, and this function is defined for every X satisfying $cond(X)$.

Definition 5 (Constraint-based Kripke Structure) *Let* $\mathcal{D}$ *be a constraint interpretation. A* constraint-based Kripke structure $\mathcal{K}$*, is a 4-tuple* $\langle \mathcal{D}, I, R, L\rangle$ *where:*

1. The carrier D *of the interpretation* $\mathcal{D}$ *is the (possibly infinite) set of* states *of the Kripke structure* $\mathcal{K}$.

2. The set $I \subseteq D$ *of* initial states *is defined by a constraint* $init(X)$*, that is, for all states* $s \in D$*, we have that:* $s \in I$ *iff* $\mathcal{D} \models init(s)$.

3. The transition relation $R \subseteq D \times D$ *is defined by a finite, exhaustive disjunction* $t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ *of events, that is, for all states* s_1 *and* s_2 *in* D*, we have that:* $s_1\, R\, s_2$ *iff* $\mathcal{D} \models t_1(s_1, s_2) \vee \ldots \vee t_k(s_1, s_2)$.

4. The function $L : D \to \mathcal{P}(Elem)$, where Elem is the set of elementary properties, is defined by associating a constraint $e(X)$ with each elementary property e, that is, for all states $s \in D$, we have that: $e \in L(s)$ iff $\mathcal{D} \models e(s)$ (note that, we use the same symbol for an elementary property and the constraint associated with it). In particular, (i) with the elementary property true we associate the constraint true, (ii) with the elementary property false we associate the constraint false, and (iii) with the elementary property init we associate the constraint init(X).

The fact that the disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ is exhaustive, together with Condition 1 of Definition 4, implies that the transition relation R is a total relation on D (and this is in accordance with Point 3 of Definition 2).

Obviously, a constraint-based Kripke structure $\langle \mathcal{D}, I, R, L \rangle$ is a Kripke structure $\langle S, I, R, L \rangle$, where S is the carrier of $\mathcal{D}$. Thus, for constraint-based Kripke structures, we use the notion of satisfaction relation given in Definition 3.

Note that *not all* reactive systems can be specified in a natural way by a constraint-based Kripke structure. In particular, Point 3 of Definition 5 implies that each state has a finite set of successor states. For instance, let us consider a system such that: (i) the set of states is the set of the natural numbers and (ii) the system has a transition from a number n to any number m greater than n, that is, the transition relation is the relation $m > n$. Let $\mathcal{N}$ be the constraint interpretation consisting of the set of natural numbers with the $>$ relation. Then, $\mathcal{N} \not\models \forall X \, \forall Y \, \forall Z \, ((X > Y \wedge X > Z) \rightarrow Y = Z)$ and, thus, the system cannot be specified by a constraint-based Kripke structure $\langle \mathcal{N}, I, R, L \rangle$ in a straightforward way. However, we will show in Section 6 that several interesting reactive systems can indeed be naturally specified by using our method.

In order to verify the temporal properties of the reactive system specified by a Kripke structure $\mathcal{K}$, we construct a locally stratified CLP program $P_{\mathcal{K}}$ that defines a binary predicate *sat* such that, for all states s and CTL formulas φ, we have that $\mathcal{K}, s \models \varphi$ iff $sat(s, \varphi) \in M(P_{\mathcal{K}})$.

Program $P_{\mathcal{K}}$, which is given in Definition 8 below, consists of clauses associated with the initial states, the transition relation, the elementary properties, the logical connectives, and the temporal operators. All these clauses are self-explanatory, except those introduced for the temporal operator AF, which require some explanation. The clauses of $P_{\mathcal{K}}$ that define the operator AF are the following:

$sat(X, af(F)) \leftarrow sat(X, F)$
$sat(X, af(F)) \leftarrow ts(X, Ys) \wedge sat\_all(Ys, af(F))$
$sat\_all([\,], F) \leftarrow$
$sat\_all([X|Xs], F) \leftarrow sat(X, F) \wedge sat\_all(Xs, F)$

In these four clauses the function symbol *af* denotes the CTL operator AF. In general, when writing an occurrence of a CTL formula as an argument of *sat*, we will slightly depart from the CTL syntax given in Section 2.3 and, according to the usual logic programming syntax, we will use lower-case function symbols instead of the corresponding upper-case operator symbols.

The meaning of the binary predicate *ts* is the following: $ts(X, Ys)$ holds iff Ys is a list of all successor states of X, that is, a state Y belongs to the list Ys iff there exists an event such that $t_i(X,Y)$ holds. Thus, the second clause for $sat(X, af(F))$ means that $sat(X, af(F))$ holds if, for all successor states Y of X, $sat(Y, af(F))$ holds.

In the case where the Kripke structure has a finite set of states, the definition of the predicate *ts* can be given by a finite collection of ground facts of the form $ts(s, [s_1, \ldots, s_q]) \leftarrow$, where $s_1, \ldots, s_q$ are the successor states of s. In the case where the Kripke structure has an infinite

set of states the definition of ts is more complicated. Indeed, the definition of ts would consist of an infinite set of ground facts of the form $ts(s, [s_1, \ldots, s_q]) \leftarrow$, even though by Point 3 of Definition 5, each state has a finite set of successor states.

We will avoid the difficulty of having an infinite set of facts in our programs, by assuming that the constraint interpretation $\mathcal{D}$ used to specify the Kripke structure, satisfies a property called *Partitioning Property* (see below). When that property holds we are able to find a finite number m of mutually exclusive constraints $c_1(X), \ldots, c_m(X)$ which determine m subsets of the set D of states such that each state s in the i-th subset of D: (i) satisfies the constraint $c_i(s)$ and does not satisfy any other constraint $c_j(s)$ with $i \neq j$, and (ii) has q_i successor states that can be determined by using q_i actions $act_{i1}(X, Y_1), \ldots, act_{iq_i}(X, Y_{q_i})$. Thus, ts can be defined by m clauses of the form:

$ts(X, Ys) \leftarrow Ys = [Y_1, \ldots, Y_{q_i}] \wedge c_i(X) \wedge act_{i1}(X, Y_1) \wedge \ldots \wedge act_{iq_i}(X, Y_{q_i})$

for $i = 1, \ldots, m$.

Now we introduce a formal definition of the Partitioning Property.

Partitioning Property
For every constraint $c(X)$, there exist m constraints $c_1(X), \ldots, c_m(X)$ such that:
(i) $\mathcal{D} \models \forall X\, (\neg c(X) \leftrightarrow (c_1(X) \vee \ldots \vee c_m(X)))$, and
(ii) for any two distinct i and j in $\{1, \ldots, m\}$, $c_i(X) \wedge c_j(X)$ is unsatisfiable, that is, $\mathcal{D} \models \neg \exists X\, (c_i(X) \wedge c_j(X))$.

The Partitioning Property tells us that for every constraint $c(X)$, the negated constraint $\neg c(X)$ is equivalent to a *finite* disjunction $c_1(X) \vee \ldots \vee c_m(X)$ of pairwise mutually exclusive constraints. If the Partitioning Property holds, we say that $\neg c(X)$ is *partitioned* into $c_1(X) \vee \ldots \vee c_m(X)$, or equivalently, $c_1(X) \vee \ldots \vee c_m(X)$ is a *partition* of the negated constraint $\neg c(X)$.

Example 1. Let us consider the set Lin_k of constraints consisting of conjunctions of linear inequations, constructed by using : (i) the predicate symbols $<$ and $\leq$, (ii) k variables, and (iii) integer coefficients. In linear inequations we can also use the symbols $>$ and $\geq$, defined as usual in terms of $<$ and $\leq$. An equation of the form $c_1 = c_2$ will be considered as an abbreviation for the conjunction of the two inequations $c_1 \leq c_2$ and $c_1 \geq c_2$. By $\mathcal{Q}_{Lin}$ we denote the constraint interpretation where: (1) the carrier is the set $\mathbb{Q}$ of rational numbers, (2) the function symbols '$+$' and '$\cdot$' are interpreted as addition and multiplication over $\mathbb{Q}$, respectively, and (3) the predicate symbols $<$ and $\leq$ are interpreted as the 'less-than' and 'less-than-or-equal-to' relations over $\mathbb{Q}$, respectively. Without loss of generality, we assume that no existentially quantified variable occurs in Lin_k, because all quantified variables can be eliminated by applying, for instance, the Fourier-Motzkin algorithm (see [5] for a proof-theoretic presentation).

The negation of any constraint in Lin_k can be partitioned into a finite disjunction of constraints, because:

(i) $\mathcal{Q}_{Lin} \models \forall\, (\neg\, (t_1 < t_2) \;\leftrightarrow\; (t_1 \geq t_2))$
(ii) $\mathcal{Q}_{Lin} \models \forall\, (\neg\, (t_1 \leq t_2) \;\leftrightarrow\; (t_1 > t_2))$

where t_1 and t_2 are linear polynomials. Thus, $\mathcal{Q}_{Lin}$ satisfies the Partitioning Property.

Let us now consider the set *TermEqs* of constraints consisting of equations between terms that are built out of an infinite set of function symbols. Let $\mathcal{H}$ be the usual interpretation, where equality is interpreted as identity between ground terms. In *TermEqs* there are constraints whose negation cannot be partitioned into a finite disjunction of constraints. For instance, the negation of the constraint $X = a$, where a is a ground term, can only be expressed by an *infinite* disjunction of constraints, as follows:

$\mathcal{H} \models \forall X\,(\neg\, X = a \leftrightarrow \bigvee_{t \in G-\{a\}} X = t)$

where G denotes the infinite set of all ground terms. Thus, $\mathcal{H}$ does *not* satisfy the Partitioning Property.

However, if we consider a set of terms constructed from a *finite* set of function symbols, then the negation of any constraint can be partitioned into a finite disjunction of constraints. For instance, if the function symbols are 0 (nullary) and s (unary), the negation of $X = s(0)$ can be partitioned into the disjunction $X = 0 \vee \exists Y\, X = s(s(Y))$. □

The construction of the definition of the binary predicate ts may require a preparatory step by which, from the disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ of events that defines the transition relation of the Kripke structure $\mathcal{K}$ (see Definition 5), we derive an equivalent disjunction $r_1(X,Y) \vee \ldots \vee r_n(X,Y)$ of *mutually exclusive, nondeterministic events* defined as follows.

Definition 6 (Nondeterministic Event) *(i) A* nondeterministic event *is a formula of the form:*

$cond(X) \wedge (act_1(X,Y) \vee \ldots \vee act_q(X,Y))$

where, for $i = 1, \ldots, q$, $cond(X) \wedge act_i(X,Y)$ *is an event.* $cond(X)$ *is the* enabling condition *of the nondeterministic event.*
(ii) A disjunction $r_1(X,Y) \vee \ldots \vee r_n(X,Y)$ *of* n *nondeterministic events is* exhaustive *if* $\mathcal{D} \models \forall X\,(cond_1(X) \vee \ldots \vee cond_n(X))$, *where, for* $i = 1, \ldots, n$, $cond_i(X)$ *is the enabling condition of* $r_i(X,Y)$.
(iii) Two nondeterministic events with enabling conditions $cond_1(X)$ *and* $cond_2(X)$, *respectively, are* mutually exclusive *if* $\mathcal{D} \models \neg\exists X\,(cond_1(X) \wedge cond_2(X))$.

If $\langle \mathcal{D}, I, R, L \rangle$ is a constraint-based Kripke structure such that $\mathcal{D}$ satisfies the Partitioning Property, then from any disjunction of events we can derive an equivalent disjunction of pairwise mutually exclusive, nondeterministic events by using the following algorithm, called *Mutex* (short for *Mut*ually *ex*clusive).

Algorithm *Mutex*
Input: an exhaustive disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ of events.
Output: an exhaustive disjunction $r(X,Y)$ of pairwise mutually exclusive, nondeterministic events such that $\mathcal{D} \models \forall X \forall Y (t_1(X,Y) \vee \ldots \vee t_k(X,Y) \leftrightarrow r(X,Y))$.

For $h = 1, \ldots, k$, let the event $t_h(X,Y)$ be of the form: $cond_h(X) \wedge act_h(X,Y)$ and let $D_h(X)$ be a partition of $\neg cond_h(X)$. First we construct the set C of constraints as follows:

$$C = \{v_1(X) \wedge \ldots \wedge v_k(X) \mid \text{for } h = 1, \ldots, k, \\ v_h(X) \text{ is either } cond_h(X) \text{ or a disjunct in } D_h(X)\}$$

Let $c_1(X), \ldots, c_n(X)$ be the satisfiable constraints in C. For $i = 1, \ldots, n$, we define the nondeterministic event $r_i(X,Y) =_{def} c_i(X) \wedge (act_{i1}(X,Y) \vee \ldots \vee act_{iq_i}(X,Y))$, where $\{act_{i1}(X,Y), \ldots, act_{iq_i}(X,Y)\} = \{act_h(X,Y) \mid$ (i) $1 \leq h \leq k$, (ii) $t_h(X,Y)$ is the event $cond_h(X) \wedge act_h(X,Y)$, and (iii) $\mathcal{D} \models \forall X(c_i(X) \rightarrow cond_h(X))\}$.
We return $r(X,Y) =_{def} r_1(X,Y) \vee \ldots \vee r_n(X,Y)$.

In Example 2 below we will see the *Mutex* algorithm in action.

Note that the construction of the set C of constraints performed by the *Mutex* algorithm may require, in the worst case, $O(m^k)$ steps, where m is the maximum number of disjuncts which

constitute a partition D_h of a negated enabling condition $\neg cond_h(X)$, for $h = 1, \ldots, k$. However, the set $\{c_1(X), \ldots, c_n(X)\}$ can be generated in less than $O(m^k)$ steps if during its construction, we eliminate the constraints which are unsatisfiable, and we replace each constraint $d(X)$ by the new constraint $solve(d(X), \{X\})$.

The correctness of the *Mutex* algorithm, stated by the following Proposition 3.1, is a straightforward consequence of the fact that the constraints in the set C are, by construction, pairwise mutually exclusive and exhaustive.

Proposition 3.1 (Correctness of the Mutex Algorithm) *Let $\mathcal{D}$ be a constraint interpretation that satisfies the Partitioning Property. From any exhaustive disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ of events defined by constraints in $\mathcal{D}$, the* Mutex *algorithm derives an exhaustive disjunction $r_1(X,Y) \vee \ldots \vee r_n(X,Y)$ of pairwise mutually exclusive, nondeterministic events such that:*

$$\mathcal{D} \models \forall X \forall Y (t_1(X,Y) \vee \ldots \vee t_k(X,Y)) \leftrightarrow (r_1(X,Y) \vee \ldots \vee r_n(X,Y))$$

Now we are able to introduce the clauses defining the predicate ts that occurs in the clause $sat(X, af(F)) \leftarrow ts(X, Ys) \wedge sat\_all(Ys, af(F))$, which will be used in the encoding of the temporal operator AF (see Definition 8 below).

Definition 7. *Let $\mathcal{K} = \langle \mathcal{D}, I, R, L \rangle$ be a constraint-based Kripke structure, where $\mathcal{D}$ satisfies the Partitioning Property. Let the transition relation R of $\mathcal{K}$ be defined by an exhaustive disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ of events and let $r_1(X,Y) \vee \ldots \vee r_n(X,Y)$ be the exhaustive disjunction of pairwise mutually exclusive, nondeterministic events constructed by the Mutex algorithm. We denote by $\widetilde{R}$ the relation defined by the disjunction $ts_1(X, Ys) \vee \ldots \vee ts_n(X, Ys)$, where, for $i = 1, \ldots, n$,*
(1) $ts_i(X, Ys) =_{def} Ys = [Y_1, \ldots, Y_{q_i}] \wedge c_i(X) \wedge act_{i1}(X, Y_1) \wedge \ldots \wedge act_{iq_i}(X, Y_{q_i})$,
(2) $Y_1, \ldots, Y_{q_i}$ are distinct variables, and
(3) $c_i(X) \wedge (act_{i1}(X,Y) \vee \ldots \vee act_{iq_i}(X,Y))$ is the nondeterministic event $r_i(X,Y)$.
The predicate ts associated with $\widetilde{R}$ *is defined by the following clauses:*

$ts(X, Ys) \leftarrow ts_1(X, Ys)$
$\ldots$
$ts(X, Ys) \leftarrow ts_n(X, Ys)$

In the following example we show how, given an exhaustive disjunction of events, the *Mutex* algorithm produces an equivalent exhaustive disjunction of mutually exclusive, nondeterministic events, and how the associated predicate ts is defined.

Example 2. Let us consider a constraint-based Kripke structure $\mathcal{K}$ and let its transition relation R be defined by the disjunction $t(X,Y)$ of the following two events $t_1(X,Y)$ and $t_2(X,Y)$:

event		enabling condition		action
$t_1(X,Y)$	$=_{def}$	$X \geq 0$	$\wedge$	$Y = X+1$
$t_2(X,Y)$	$=_{def}$	$X \leq 1$	$\wedge$	$Y = X-1$

where $\geq$ and $\leq$ denote the usual inequality relations between rational numbers. Since the negation of the constraints $X \geq 0$ and $X \leq 1$ can be expressed as the constraints $X < 0$ and $X > 1$, respectively, by using the *Mutex* algorithm we can rewrite $t(X,Y)$ as the disjunction of the following three nondeterministic events:

event		enabling condition		action
$r_1(X,Y)$	$=_{def}$	$X<0$	$\wedge$	$Y=X-1$
$r_2(X,Y)$	$=_{def}$	$X\geq 0 \wedge X\leq 1$	$\wedge$	$(Y=X+1 \vee Y=X-1)$
$r_3(X,Y)$	$=_{def}$	$X>1$	$\wedge$	$Y=X+1$

Thus, the predicate ts is defined by the following three clauses:

$ts(X, Ys) \leftarrow Ys = [Y] \wedge X<0 \wedge Y=X-1$
$ts(X, Ys) \leftarrow Ys = [Y_1, Y_2] \wedge X\geq 0 \wedge X\leq 1 \wedge Y_1=X+1 \wedge Y_2=X-1$
$ts(X, Ys) \leftarrow Ys = [Y] \wedge X>1 \wedge Y=X+1$ □

The following proposition states the correctness of the program that defines the predicate ts constructed from the transition relation R as indicated in Definition 7.

Proposition 3.2. *Let Ts be the set of clauses defining the predicate ts associated with the transition relation $\widetilde{R}$. For all $s \in D$, $\{s_1, \ldots, s_q\} = \{s' \in D \mid s\, R\, s'\}$ iff $M(Ts) \models ts(s, ss)$ for some list ss which is a permutation of $[s_1, \ldots, s_q]$.*

Now we introduce the locally stratified constraint logic program $P_{\mathcal{K}}$ that encodes a Kripke structure $\mathcal{K}$ and the satisfiability of a CTL formula in $\mathcal{K}$. As already mentioned, when writing an occurrence of a CTL formula as an argument of sat, we will use lower-case function symbols instead of the corresponding upper-case operator symbols.

Definition 8 (Encoding Program) *Given a constraint-based Kripke structure $\mathcal{K}=\langle \mathcal{D}, I, R, L\rangle$ such that $\mathcal{D}$ satisfies the Partitioning Property, the* encoding program $P_{\mathcal{K}}$ for $\mathcal{K}$ *consists of the following clauses:*

$t(X,Y) \leftarrow t_1(X,Y)$
$\ldots$
$t(X,Y) \leftarrow t_k(X,Y)$

$ts(X, Ys) \leftarrow ts_1(X, Ys)$
$\ldots$
$ts(X, Ys) \leftarrow ts_n(X, Ys)$

$sat(X, e_1) \leftarrow e_1(X)$
$\ldots$
$sat(X, e_m) \leftarrow e_m(X)$

$sat(X, \neg F) \leftarrow \neg sat(X, F)$
$sat(X, F_1 \wedge F_2) \leftarrow sat(X, F_1) \wedge sat(X, F_2)$

$sat(X, ex(F)) \leftarrow t(X,Y) \wedge sat(Y, F)$

$sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_2)$
$sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_1) \wedge t(X,Y) \wedge sat(Y, eu(F_1, F_2))$

$sat(X, af(F)) \leftarrow sat(X, F)$
$sat(X, af(F)) \leftarrow ts(X, Ys) \wedge sat\_all(Ys, af(F))$

$sat\_all([\,], F) \leftarrow$
$sat\_all([X|Xs], F) \leftarrow sat(X, F) \wedge sat\_all(Xs, F)$

where:
(1) $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ is the disjunction of constraints that defines the transition relation R;

(2) $ts_1(X, Ys) \vee \ldots \vee ts_n(X, Ys)$ *is the disjunction that defines the relation* $\widetilde{R}$*;*
(3) for $i = 1, \ldots, m$, $e_i(X)$ *is the constraint associated with the elementary property* e_i*. In particular, (i) if* e_i *is the elementary property true, then* $e_i(X)$ *is the constraint true, (ii) if* e_i *is the elementary property false, then* $e_i(X)$ *is the constraint false, and (iii) if* e_i *is the elementary property init, then* $e_i(X)$ *is the constraint* $init(X)$ *that defines the set* I *of initial states. For reasons of simplicity, we usually omit to write the clause* $sat(X, false) \leftarrow false$ *(indeed, this clause can be removed from* $P_{\mathcal{K}}$ *by applying the* clause removal rule *R6 which will be presented in Section 4).*

As already mentioned in Section 2.3, the CTL formulas written by using the operators EF, EG, AX, AU, and AG are considered as abbreviations of formulas written by using EX, EU, and AF. Thus, in order to verify formulas that use EF, EG, AX, AU, and AG, we can rewrite them as formulas that use EX, EU, and AF only, and then we can use the encoding program of Definition 8.

Since the operator EF is often used in our examples, for the reader's convenience, we present specialized clauses for dealing with formulas of the form $EF(\varphi)$. Since $EF(\varphi)$ stands for $EU(true, \varphi)$ and $sat(X, true)$ is true for all states X, the clauses for $EF(\varphi)$ can be derived as a specialization of the clauses:

$sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_2)$
$sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_1) \wedge t(X, Y) \wedge sat(Y, eu(F_1, F_2))$

by: (i) substituting $true$ for F_1, (ii) substituting $ef(F_2)$ for $eu(true, F_2)$, and (iii) removing the occurrence of $sat(X, true)$. Thus, we obtain the following two clauses:

$sat(X, ef(F)) \leftarrow sat(X, F)$
$sat(X, ef(F)) \leftarrow t(X, Y) \wedge sat(Y, ef(F))$

The program $P_{\mathcal{K}}$ constructed as indicated in Definition 8 is locally stratified w.r.t. the stratification function σ defined as follows:

$\sigma(t(s_1, s_2)) = 0$, for all ground terms s_1 and s_2,
$\sigma(ts(s, ss)) = 0$, for all ground terms s and ss,
$\sigma(sat(s, \varphi)) = size(\varphi)$, for all ground terms s and φ,
$\sigma(sat\_all(s, \varphi)) = size(\varphi)$, for all ground terms s and φ,

where, for any ground term t, $size(t)$ is the number of occurrences of function symbols in t.

The following result, whose proof is given in the Appendix, establishes the correctness of the encoding of a Kripke structure.

Theorem 3.3 (Correctness of the Encoding Program) *Let* $\mathcal{K} = \langle \mathcal{D}, I, R, L \rangle$ *be a constraint-based Kripke structure and let* $P_{\mathcal{K}}$ *be the encoding program for* $\mathcal{K}$*. For all states* $s \in D$ *and CTL formulas* φ*, we have that:*

$$\mathcal{K}, s \models \varphi \quad \textit{iff} \quad sat(s, \varphi) \in M(P_{\mathcal{K}})$$

In the following example we consider a simple reactive system specified by a Kripke structure $\mathcal{K}$ based on a constraint interpretation $\mathcal{D}$ and we construct the corresponding program $P_{\mathcal{K}}$.

Example 3. Let us consider the reactive system depicted in Figure 1. A state of this system is a $\langle$*control state*, *counter*$\rangle$ pair. The control state is either a or b and the counter is a rational number. The constraint-based Kripke structure $\mathcal{K} = \langle \mathcal{D}, I, R, L \rangle$ that models that system can be defined as follows.

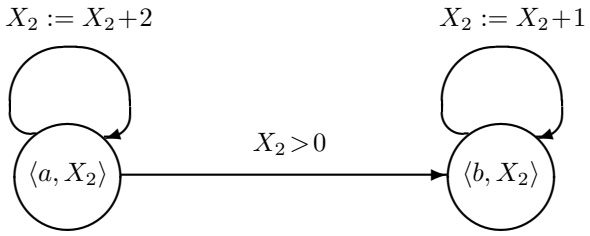


Figure 1: A simple reactive system. The initial state is $\langle a, 0\rangle$.

The carrier of $\mathcal{D}$ is the set $D = \{a, b\} \times \mathbb{Q}$, where $\mathbb{Q}$ is the set of rational numbers. We assume that in $\mathcal{D}$ the following operations and relations are defined: (i) addition of rational numbers, (ii) equality between elements in $\{a, b\}$, and (iii) equality and inequality between rational numbers. We use the same symbol $=$ to denote the equality between elements in $\{a, b\}$ and the equality between rational numbers.

The set I of initial states is defined by the constraint $init(\langle X_1, X_2\rangle) =_{def} (X_1 = a \wedge X_2 = 0)$, that is, I is the singleton $\{\langle a, 0\rangle\}$.

The transition relation R from state $\langle X_1, X_2\rangle$ to state $\langle Y_1, Y_2\rangle$ is defined as the disjunction of the following three events:

event		enabling condition		action
$t_1(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = a$	$\wedge$	$Y_1 = a \wedge Y_2 = X_2 + 2$
$t_2(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = a \wedge X_2 > 0$	$\wedge$	$Y_1 = b \wedge Y_2 = X_2$
$t_3(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = b$	$\wedge$	$Y_1 = b \wedge Y_2 = X_2 + 1$

Let the elementary property *neg* hold in a state $\langle X_1, X_2\rangle$ iff $X_2 < 0$. Note that the conditions occurring in the events $t_1(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$ and $t_2(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$ are not mutually exclusive because $\mathcal{D} \models \exists X_1 \exists X_2\, ((X_1 = a) \wedge (X_1 = a \wedge X_2 > 0))$. Thus, in order to construct the clauses for the predicate ts we have to apply the *Mutex* algorithm. This algorithm can indeed be applied because the constraint interpretation $\mathcal{D}$ satisfies the Partitioning Property (see also Example 1). In particular, we use the following equivalences:

$\mathcal{D} \models \forall X((\neg X > 0) \leftrightarrow X \leq 0)$
$\mathcal{D} \models \forall X((\neg X = a) \leftrightarrow X = b)$

From the transition relation R the *Mutex* algorithm derives the following three, pairwise mutually exclusive, nondeterministic events:

nondeterministic event		enabling condition		action
$r_1(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = a \wedge X_2 \leq 0$	$\wedge$	$Y_1 = a \wedge Y_2 = X_2 + 2$
$r_2(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = a \wedge X_2 > 0$	$\wedge$	$((Y_1 = a \wedge Y_2 = X_2 + 2) \vee (Y_1 = b \wedge Y_2 = X_2))$
$r_3(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle)$	$=_{def}$	$X_1 = b$	$\wedge$	$Y_1 = b \wedge Y_2 = X_2 + 1$

The encoding program $P_{\mathcal{K}}$ consists of the following clauses:

$t(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle) \leftarrow X_1 = a \wedge Y_1 = a \wedge Y_2 = X_2 + 2$
$t(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_1 = b \wedge Y_2 = X_2$
$t(\langle X_1, X_2\rangle, \langle Y_1, Y_2\rangle) \leftarrow X_1 = b \wedge Y_1 = b \wedge Y_2 = X_2 + 1$

$ts(\langle X_1, X_2\rangle, [\langle Y_1, Y_2\rangle]) \leftarrow X_1 = a \wedge X_2 \leq 0 \wedge Y_1 = a \wedge Y_2 = X_2 + 2$
$ts(\langle X_1, X_2\rangle, [\langle Y_{11}, Y_{12}\rangle, \langle Y_{21}, Y_{22}\rangle]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_{11} = a \wedge Y_{12} = X_2 + 2 \wedge$
$\qquad Y_{21} = b \wedge Y_{22} = X_2$
$ts(\langle X_1, X_2\rangle, [\langle Y_1, Y_2\rangle]) \leftarrow X_1 = b \wedge Y_1 = b \wedge Y_2 = X_2 + 1$

$sat(\langle X_1, X_2\rangle, init) \leftarrow X_1 = a \wedge X_2 = 0$
$sat(\langle X_1, X_2\rangle, true) \leftarrow$
$sat(\langle X_1, X_2\rangle, neg) \leftarrow X_2 < 0$

together with the clauses defining $sat(X, \neg F)$, $sat(X, \neg F_1 \wedge F_2)$, $sat(X, ex(F))$, $sat(X, eu(F_1, F_2))$, $sat(X, af(F))$, and $sat\_all(Xs, F)$ (see Definition 8). □

The problem of proving $\mathcal{K}, s \models \varphi$, for all initial states $s \in I$, can be encoded as a goal for a CLP program as specified by the following definition.

Definition 9. *Let $\mathcal{K}$ be a Kripke structure, let $P_{\mathcal{K}}$ be the encoding program for $\mathcal{K}$, and let φ be a CTL formula. By $P_{\mathcal{K}}[\varphi]$ we denote the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, where γ_1 and γ_2 are the following two clauses:*

γ_1 : $prop \leftarrow \neg negprop$
γ_2 : $negprop \leftarrow sat(X, init \wedge \neg\varphi)$

The following result is a direct consequence of Theorem 3.3 and of the definition of perfect model, and it is the basis of our verification technique.

Corollary 3.4. *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,*

$$\mathcal{K}, s \models \varphi, \ \textit{for all states } s \in I \quad \textit{iff} \quad prop \in M(P_{\mathcal{K}}[\varphi]).$$

4. The Rules for Specializing CLP Programs

The process of specializing a given program P and deriving a new program P_{sp}, which is required by our verification task, will be formalized as the construction of a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where: (i) $P_0 = P$, (ii) $P_n = P_{sp}$ and, (iii) for $k = 0, \ldots, n$–1, program P_{k+1} is obtained from program P_k by applying one of the transformation rules listed below. These rules are variants, tailored to the verification technique we present in this paper, of the *unfold/fold rules* considered in the literature for transforming logic programs and constraint logic programs (see, in particular, [8, 23, 26, 42, 61, 66]).

The first rule R1 allows us to introduce a new predicate definition by adding to program P_k, for some k such that $0 \leq k \leq n-1$, a new clause whose body consists of a constrained atom.

Rule R1 (*Constrained Atomic Definition*)
By *constrained atomic definition* (or *definition*, for short), we introduce a clause, called a *definition*, of the form

δ : $newp(X_1, \ldots, X_m) \leftarrow c \wedge A$

where: (i) *newp* is a predicate symbol not occurring in the transformation sequence $P_0, \ldots, P_k$, (ii) $FV(A) = \{X_1, \ldots, X_m\}$, (iii) $FV(c) \subseteq \{X_1, \ldots, X_m\}$, and (iv) the predicate of A occurs in P_0. We derive the program $P_{k+1} = P_k \cup \{\delta\}$.
For $k \geq 0$, $Defs_k$ denotes the set of definitions introduced during the construction of the transformation sequence $P_0, \ldots, P_k$. In particular, $Defs_0 = \emptyset$.

Each of the following two *unfolding* rules R2 and R3 corresponds to a symbolic computation step. These unfolding rules essentially consist in replacing an atom A occurring in the body of

a clause of P_k by the bodies of the clauses which define A in program P_k. If the atom A to be replaced occurs positively in the body of the clause to be unfolded, we apply the *positive unfolding* rule R2. If A occurs negatively, we apply the *negative unfolding* rule R3.

Rule R2 (*Positive Unfolding*)
Let $\gamma:\ H \leftarrow c \wedge G_L \wedge A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

$$\begin{array}{ll} \gamma_1: & K_1 \leftarrow c_1 \wedge G_1 \\ & \ldots \\ \gamma_m: & K_m \leftarrow c_m \wedge G_m \end{array}$$

where $m \geq 0$, be all clauses of program P'_k such that, for $i = 1, \ldots, m$, the constraint $c \wedge A = K_i \wedge c_i$ is satisfiable.
By *unfolding* γ *w.r.t.* A we derive the clauses

$$\begin{array}{ll} \eta_1: & H \leftarrow c \wedge A = K_1 \wedge c_1 \wedge G_L \wedge G_1 \wedge G_R \\ & \ldots \\ \eta_m: & H \leftarrow c \wedge A = K_m \wedge c_m \wedge G_L \wedge G_m \wedge G_R \end{array}$$

and we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$.

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

By using rule R3, we can unfold a clause w.r.t. a negative literal provided that this literal is a *decided literal* in the sense specified by the following definition.

Definition 10 (Decided Predicate and Decided Literal) *We say that a predicate p is* decided *in a program P if $Def(p, P)$ is a (possibly empty) set of constrained facts. A literal L is* decided *in P if the predicate of L is decided in P.*

We will see in Section 6 that our restricted form of negative unfolding w.r.t. decided literals is sufficient for many interesting verification examples.

When we apply the negative unfolding rule R3 we assume that the Partition Property holds for the class of constraints we consider.

Rule R3 (*Negative Unfolding*)
Let $\gamma:\ H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let us assume that $\neg A$ is a decided literal in P'_k and let

$$\begin{array}{ll} \gamma_1: & K_1 \leftarrow c_1 \\ & \ldots \\ \gamma_m: & K_m \leftarrow c_m \end{array}$$

where $m \geq 0$, be all clauses of program P'_k such that, for $i = 1, \ldots, m$, the constraint $c \wedge A = K_i \wedge c_i$ is satisfiable.
By *unfolding* γ *w.r.t.* $\neg A$ we derive the clauses

$$\begin{array}{ll} \eta_1: & H \leftarrow c \wedge f_1 \wedge G_L \wedge G_R \\ & \ldots \\ \eta_s: & H \leftarrow c \wedge f_s \wedge G_L \wedge G_R \end{array}$$

where $s \geq 0$ and $f_1, \ldots, f_s$ are constraints obtained by performing the following four steps.
Step 1. (*Project*) We consider the following formula ψ_0, which is equivalent to $\neg A$ in $M(P_k)$:

$$\psi_0:\ \neg(\exists Y_1\,(A = K_1 \wedge c_1) \vee \ldots \vee \exists Y_m\,(A = K_m \wedge c_m))$$

where, for $i = 1, \ldots, m$, $Y_i = FV(K_i) \cup FV(c_i)$. From ψ_0 we derive the following equivalent formula:

$\psi_1 : \ \neg(d_1 \vee \ldots \vee d_m)$

where, for $i = 1, \ldots, m$, $d_i = solve(A{=}K_i \wedge c_i, FV(A))$.

Step 2. (*Push* $\neg$ *inside*) We apply De Morgan's law to ψ_1 and we derive the following equivalent formula:

$\psi_2 : \ \neg d_1 \wedge \ldots \wedge \neg d_m$

Step 3. (*Eliminate* $\neg$) From ψ_2 we derive an equivalent formula of the form:

$\psi_3 : \ D_1 \wedge \ldots \wedge D_m$

where, for $i = 1, \ldots, m$, D_i is a partition of the negated constraint $\neg d_i$ (that is, a disjunction of constraints equivalent to $\neg d_i$). This step is possible because the Partition Property holds.

Step 4. (*Push* $\vee$ *outside*) We apply to ψ_3 as long as possible the following rewriting of formulas (that is, the distributivity law):

$\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \ \longrightarrow \ (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

and we get an equivalent formula of the form:

$\psi_4 : \ f_1 \vee \ldots \vee f_s$

Note that for $i = 1, \ldots, s$, f_i is a constraint of the form $e_1 \wedge \ldots \wedge e_m$, where, for $j = 1, \ldots, m$, e_j is a disjunct occurring in D_j.

We derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_s\}$.

Note that: (i) if $m = 0$, that is, if we unfold clause γ w.r.t. a negative literal $\neg A$ such that the constraint $c \wedge A{=}K_i \wedge c_i$ is satisfiable for no clause in P'_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause γ, and (ii) if we unfold clause γ w.r.t. a negative literal $\neg A$ such that a variant of the fact $A \leftarrow$ is in P'_k, then we derive the new program P_{k+1} by deleting clause γ from P_k.

The following *folding* rules R4 and R5 allow us to replace an atom A which is an instance of the body of a definition by the corresponding instance of the head of the definition. If the atom A occurs positively in the body of a clause, then we apply the *positive folding* rule R4, otherwise, if A occurs negatively in the body of a clause, then we apply the *negative folding* rule R5.

Rule R4 (*Positive Folding*)
Let $\gamma : \ H \leftarrow c \wedge G_L \wedge A \wedge G_R$ be a clause in P_k and let $\delta : \ N \leftarrow d \wedge B$ be a clause in $Defs_k$ such that: (i) δ has no variables in common with γ, (ii) $A = B\vartheta$, for some substitution ϑ, and (iii) $\mathcal{D} \models \forall\, (c \rightarrow \exists Y\, d\vartheta)$, where $Y = FV(d) - FV(N)$.
By *folding* γ *w.r.t.* A *using* δ we derive the clause

$\eta : \ H \leftarrow c \wedge G_L \wedge N\vartheta \wedge G_R$

and we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

Rule R5 (*Negative Folding*)
Let $\gamma : \ H \leftarrow c \wedge G_1 \wedge \neg A \wedge G_2$ be a clause in P_k and let $\delta : \ N \leftarrow d \wedge B$ be a clause in $Defs_k$ such that: (i) δ has no variables in common with γ, (ii) $A = B\vartheta$, for some substitution ϑ, and (iii) $\mathcal{D} \models \forall\, (c \rightarrow d\vartheta)$.
By *folding* γ *w.r.t.* $\neg A$ *using* δ we derive the clause

$\eta : \ H \leftarrow c \wedge G_L \wedge \neg N\vartheta \wedge G_R$

and we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

The following *clause removal* rule R6 can be used for removing from P_k a redundant clause γ, that is, a clause such that $M(P_k) = M(P_k - \{\gamma\})$. Since the problem of testing whether or not $M(P_k) = M(P_k - \{\gamma\})$ is undecidable, we will consider some sufficient conditions based on decidable properties. These sufficient conditions are based on the notions of *subsumed clause* and *useless clause*, which we now define.

A clause of the form $H \leftarrow c \wedge G$ is *subsumed* by a constrained fact of the form $H \leftarrow d$ if $\mathcal{D} \models \forall\, (c \rightarrow d)$.

The set of *useless predicates* in a program P is the maximal set U of predicates occurring in P such that a predicate p is in U iff every clause γ in $Def\,(p, P)$ is of the form $H \leftarrow c \wedge G_1 \wedge q(\ldots) \wedge G_2$ for some q in U. A clause in a program P is *useless* if the predicate of its head is useless in P. For example, in the following program:

$p(X) \leftarrow q(X) \wedge \neg r(X)$
$q(X) \leftarrow p(X)$
$r(X) \leftarrow X > 0$

p and q are useless predicates, while r is not useless.

Rule R6 (*Clause Removal*)
Let γ be a clause in P_k. By *clause removal* we derive the program $P_{k+1} = P_k - \{\gamma\}$ if one of the following two cases occurs:

R6s. γ is subsumed by a constrained fact occurring in $P_k - \{\gamma\}$;

R6u. γ is useless in P_k.

The following *constraint replacement* rule R7 allows us to replace a constraint occurring in the body of a clause by an equivalent constraint.

Rule R7 (*Constraint Replacement*)
Let $\gamma_1 :\ H \leftarrow c_1 \wedge G$ be a clause in P_k. Suppose that for some constraint c_2, we have that:

$\mathcal{D} \models \forall\, (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2)$

where: (i) $Y = FV(c_1) - (FV(H) \cup FV(G))$, and (ii) $Z = FV(c_2) - (FV(H) \cup FV(G))$. Then by *constraint replacement* we derive the clause

$\gamma_2 :\ H \leftarrow c_2 \wedge G$

and we derive the program $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$.

The following Theorem 4.1 states that, under suitable conditions, the transformation rules R1–R7 we have presented, preserve the perfect model semantics. These conditions ensure that during the construction of a transformation sequence, each clause introduced by the constrained atomic definition (Rule R1) and used for positive folding (Rule R4), is unfolded (before or after folding) w.r.t. the unique atom in its body.

Theorem 4.1 (Correctness of the Transformation Rules) *Let P_0 be a locally stratified program and let $P_0, \ldots, P_n$ be a transformation sequence. Let us assume that for every $k \in \{1, \ldots, n-1\}$ such that P_{k+1} is derived by applying Rule R4 and folding a clause in P_k using a clause δ in $Defs_k$, there exists $j \in \{1, \ldots, n-1\} - \{k\}$ such that δ belongs to P_j and P_{j+1} is derived by applying Rule R2 and unfolding δ w.r.t. the unique atom in its body.*
Then P_n is locally stratified and for every ground atom A whose predicate occurs in P_0, we have that $A \in M(P_0)$ iff $A \in M(P_n)$.

Theorem 4.1 is a consequence of the fact that the transformation rules R1–R7 are particular cases of the transformation rules presented in [26], and the latter rules preserve the perfect model semantics (see Theorem 3 in [26]).

5. The Specialization Strategy

In this section we present the specialization strategy which will be used for verifying CTL properties of reactive systems.

Suppose that we are given a reactive system specified by a Kripke structure $\mathcal{K}$ and a CTL formula φ. We want to verify that, for all initial states s, the formula φ holds in state s, that is, $\mathcal{K}, s \models \varphi$. This verification can be performed by considering the program $P_{\mathcal{K}}[\varphi]$ constructed as described in Definition 9. Indeed, by Corollary 3.4 of Section 3 we have that:

$$\mathcal{K}, s \models \varphi, \text{ for all initial states } s \quad \text{iff} \quad \mathit{prop} \in M(P_{\mathcal{K}}[\varphi])$$

In order to accomplish our verification task, in this section we will present a specialization strategy which applies the transformation rules of Section 4 and from program $P_{\mathcal{K}}[\varphi]$ derives a specialized program P_{sp}. The correctness of the transformation rules with respect to the perfect model semantics (see Theorem 4.1 of Section 4) ensures that $\mathit{prop} \in M(P_{\mathcal{K}}[\varphi])$ iff $\mathit{prop} \in M(P_{sp})$. Thus, if P_{sp} contains the fact $\mathit{prop} \leftarrow$, then for all initial states s, we have that $\mathcal{K}, s \models \varphi$. Moreover, if P_{sp} does not contain any clause for prop, then there exists an initial state s such that $\mathcal{K}, s \not\models \varphi$.

We will make sure that our specialization strategy always terminates by employing a suitable *generalization* technique. However, due to the undecidability of CTL for infinite state systems, we cannot hope that in all cases our strategy allows us to determine whether or not $\mathcal{K}, s \models \varphi$. Indeed, in some cases our strategy derives a program P_{sp} where prop is defined by one or more clauses with non-empty body.

5.1. Outline of the Specialization Strategy

Our strategy specializes the program $P_{\mathcal{K}}[\varphi]$ w.r.t. prop by sequentially applying the *UDF procedure* and the *RU procedure* which we will describe below. (*UDF* stands for 'unfolding–definition–folding' and *RU* stands for 'removal–unfolding'.) Now, recall that by $P_{\mathcal{K}}[\varphi]$ we denote the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, where γ_1 and γ_2 are the clauses (see Definition 9):

$\gamma_1 : \ \mathit{prop} \leftarrow \neg \mathit{negprop}$
$\gamma_2 : \ \mathit{negprop} \leftarrow \mathit{sat}(X, \mathit{init} \wedge \neg\varphi)$

The *UDF* procedure specializes the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ w.r.t. $\mathit{negprop}$ leaving clause γ_1 untouched, thereby deriving a program P_A of the form $P_{\mathit{negprop}} \cup \{\gamma_1\}$. The *RU* procedure completes the specialization of $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ w.r.t. prop by performing the specialization of P_A w.r.t. prop.

The *UDF* procedure consists in the iterated execution of the 'unfolding–definition–folding cycle' which is typical of the specialization techniques based on the unfolding/folding approach (see, for instance, [25, 53, 60]). In a generic execution of the unfolding–definition–folding cycle we consider a clause γ of the form $H \leftarrow c(X) \wedge \mathit{sat}(X, \psi)$ (at the first execution we consider clause γ_2). By applying the positive unfolding rule R2 one or more times, from clause γ we derive a new set of clauses, say Γ. Then, for each constrained literal of the form $c_1(X) \wedge \mathit{sat}(X, \psi_1)$ or $c_1(X) \wedge \neg \mathit{sat}(X, \psi_1)$ occurring in the body of a clause in Γ, by applying the definition rule R1, we introduce a new definition of the form $\mathit{newp}(X) \leftarrow d(X) \wedge \mathit{sat}(X, \psi_1)$ such that $\mathcal{D} \models \forall X \, (c_1(X) \rightarrow d(X))$. Thus, by applying the positive and negative folding rules R4 and R5, from Γ we derive a new set Φ of clauses without occurrences of *sat* literals. For each new definition introduced by Rule R1, we perform again unfolding, definition, and folding steps, and we iterate this unfolding–definition–folding cycle until no new definitions need to be introduced for applying the folding rule w.r.t. all (positive or negative) constrained occurrences of the *sat*

literals. This is the case when those occurrences can be folded using the definitions that have been already introduced by previous applications of the definition rule.

The *UDF* procedure depends on the auxiliary procedures used for unfolding and for the introduction of new definitions which will be presented in Section 5.2.1. In particular, the introduction of new definitions is realized by applying a suitable *generalization* technique which ensures that a finite set of definitions will be introduced during the application of the *UDF* procedure, thereby guaranteeing the termination of this procedure (see Theorem 5.9).

The *RU* procedure derives decided predicates, with the objective of obtaining a final program P_{sp} where *prop* is a decided predicate, that is, a program P_{sp} which either contains the fact $prop \leftarrow$ or contains no clauses for *prop*. We will prove that the program, denoted by P_A, which is derived at the end of the *UDF* procedure, is stratified, that is, P_A has a finite number of strata. Thus, the *RU* procedure works bottom-up on the strata of P_A and simplifies the definition of every predicate p occurring in the program, with the aim of deriving either the fact $p \leftarrow$ or the empty definition for p. In order to do so the *RU* procedure applies: (i) the clause removal rule R6 and (ii) the positive and negative unfolding rules R2 and R3 w.r.t. decided literals.

Before describing in more detail the *UDF* and *RU* procedures of our specialization strategy, let us illustrate through an example some of the ideas upon which our strategy is based.

Example 4. Let us consider a reactive system consisting of an integer counter X which is initialized to 1 and is incremented by 1 at each time unit. The state of the system is the value of the counter X. We want to prove that starting from the initial state it is impossible to reach a state where the value of the counter is 0.

The system is specified by the following constraint-based Kripke structure $Count = \langle \mathcal{I}, I, R, L \rangle$, where: (i) $\mathcal{I}$ is the usual constraint interpretation for the integer numbers with addition, (ii) the set I of the initial states is defined by the constraint $X = 1$, (iii) the transition relation R is defined by the single event $Y = X + 1$, (iv) the function L is defined by associating with the elementary property *null* the constraint $X = 0$.

The encoding program for *Count* is the following CLP program P_{Count} (see Definition 8):

1. $t(X, Y) \leftarrow Y = X + 1$
2. $sat(X, init) \leftarrow X = 1$
3. $sat(X, null) \leftarrow X = 0$
4. $sat(X, \neg F) \leftarrow \neg sat(X, F)$
5. $sat(X, ef(F)) \leftarrow sat(X, F)$
6. $sat(X, ef(F)) \leftarrow t(X, Y) \wedge sat(Y, ef(F))$

where we have listed only the clauses which are needed in this verification example. Our property of interest is expressed by the CTL formula $\neg EF(null)$ and we want to verify that $Count, X \models \neg EF(null)$ holds for the initial state defined by the constraint $X = 1$. In order to verify this property, we introduce the following two clauses:

$\gamma_1 :\ prop \leftarrow \neg negprop$
$\gamma_2 :\ negprop \leftarrow sat(X, init \wedge ef(null))$

where we have simplified $\neg\neg ef(null)$ to $ef(null)$. By Corollary 3.4, we can perform our verification task by proving that $prop \in M(P_{Count}[\neg EF(null)])$, where $P_{Count}[\neg EF(null)]$ denotes the program $P_{Count} \cup \{\gamma_1, \gamma_2\}$.

Let us remark that this proof cannot be done by using the standard operational semantics of constraint logic programs (see, for instance, [45]). Indeed, by using the standard operational

semantics, $P_{Count}[\neg EF(null)]$ does not terminate for the goal *prop*, because the following infinite derivation is generated from *negprop*:

$sat(X, init \wedge ef(null))$
$sat(X, init) \wedge sat(X, ef(null))$
$X = 1 \wedge sat(X, ef(null))$
$X = 1 \wedge X1 = X + 1 \wedge sat(X1, ef(null))$
$X = 1 \wedge X1 = X + 1 \wedge X2 = X1 + 1 \wedge sat(X2, ef(null))$
$\ldots$

Note that program $P_{Count}[\neg EF(null)]$ does not terminate either if one uses a system for *tabled constraint logic programming* [15], because each element of the above infinite derivation is a constrained atom which is not an instance of a preceding element.

Now we prove that $prop \in M(P_{Count}[\neg EF(null)])$ by using the transformation rules of Section 4 according to the *UDF* and *RU* procedures of the specialization strategy we have outlined above.

UDF procedure. We repeatedly perform the unfolding–definition–folding transformation cycle described above, starting from clause γ_2.

(A.1) We unfold clause γ_2 and we get:

7. $negprop \leftarrow X = 1 \wedge sat(X, ef(null))$

We introduce the following new definition:

8. $new1(X) \leftarrow X = 1 \wedge sat(X, ef(null))$

and we fold clause 7 using clause 8, thereby deriving the clause:

9. $negprop \leftarrow X = 1 \wedge new1(X)$

(A.2) The specialization process continues by applying the unfolding–definition–folding transformation cycle starting from the new definition clause 8. We unfold clause 8 and we get:

10. $new1(X) \leftarrow X = 1 \wedge Y = X + 1 \wedge sat(Y, ef(null))$

In order to fold clause 10 we introduce the following new definition:

11. $new2(X) \leftarrow X \geq 1 \wedge sat(X, ef(null))$

whose body is obtained from the body of clause 8 by *generalizing* the constraint $X = 1$ to the constraint $X \geq 1$. By applying rule R4 we fold clause 10 using clause 11 and we get:

12. $new1(X) \leftarrow X = 1 \wedge Y = X + 1 \wedge new2(Y)$

(A.3) We perform once again the unfolding–definition–folding transformation cycle starting from clause 11. We first unfold clause 11, thereby deriving the following clause:

13. $new2(X) \leftarrow X \geq 1 \wedge Y = X + 1 \wedge sat(Y, ef(null))$

No new definition is needed for folding clause 13. Indeed clause 13 can be folded by using clause 11 thereby deriving:

14. $new2(X) \leftarrow X \geq 1 \wedge Y = X + 1 \wedge new2(Y)$

This folding step concludes the *UDF* procedure. By applying this procedure we have derived the following program P_A:

γ_1. $prop \leftarrow \neg negprop$
9. $negprop \leftarrow X = 1 \wedge new1(X)$
12. $new1(X) \leftarrow X = 1 \wedge Y = X + 1 \wedge new2(Y)$
14. $new2(X) \leftarrow X \geq 1 \wedge Y = X + 1 \wedge new2(Y)$

RU procedure. We derive decided predicates as follows. The predicates *negprop*, *new*1, and *new*2 are useless in P_A (see Section 4). Thus, by using the clause removal rule R6u, clauses 9, 12, and 14 are deleted, and we derive a program consisting of clause γ_1 only. By using the negative unfolding rule R3 we unfold clause γ_1 w.r.t. *negprop* and we derive a final specialized program P_{sp} where the definition of predicate *prop* consists of the following clause only:

15. $prop \leftarrow$

Thus, $prop \in M(P_{sp})$ and, by the correctness of the rules (see Theorem 4.1 above), we get that $prop \in M(P_{Count}[\neg EF(null)])$. By Corollary 3.4 at the end of Section 3 we have that $Count, X \models \neg EF(null)$ holds for the initial state $X=1$, and this concludes our proof. □

Now we want to briefly discuss a few points related to the proof we have presented in Example 4.

(1) We use the folding rule to infer that some atoms are infinitely failed and, thus, they are false in the perfect model of the program. Indeed, by folding, the clauses defining infinitely failed atoms are transformed into useless clauses, which are then deleted by applying the clause removal rule. For instance, we infer that the atom $new2(X)$ is infinitely failed (see clause 14) by folding clause 13 by using clause 11.

(2) In order to perform the folding steps required for generating useless clauses as indicated at Point (1), we may need to introduce new definitions by applying a generalization technique. In our case we have introduced clause 11 by generalizing the body of clause 8, and indeed, by using clause 11 we were able to perform folding steps w.r.t. all constrained *sat* literals occurring in the program at hand.

(3) The choice of a suitable generalization technique plays a crucial rôle in our verification method because, as already mentioned, it ensures the termination of the specialization strategy. However, in some cases generalization can also prevent the proof of the properties of interest as indicated by the following example.

Example 5. Let us consider again the above Example 4 and suppose that we generalize the constraint $X=1$ in the body of clause 8 to *true*, instead of $X \geq 1$. Then, instead of clause 11, we would have introduced the following clause:

11*. $new3(X) \leftarrow sat(X, ef(null))$

In this case the program P_A derived at the end of the *UDF* procedure is:

γ_1. $prop \leftarrow \neg negprop$
9. $negprop \leftarrow X=1 \wedge new1(X)$
16. $new1(X) \leftarrow X=1 \wedge Y=X+1 \wedge new3(Y)$
17. $new3(X) \leftarrow X=0$
18. $new3(X) \leftarrow Y=X+1 \wedge new3(Y)$

The application of the *RU* procedure does not change program P_A because no clause is useless in P_A. Thus, no decided predicate can be derived and the property of interest cannot be proved. □

5.2. The Strategy for Specializing CLP Programs

Suppose that $\mathcal{K}$ is a Kripke structure based on a constraint interpretation $\mathcal{D}$, and φ is the CTL property to be verified. Let us consider the program $P_{\mathcal{K}}[\varphi]$ constructed as described in Definition 9 of Section 3. Our specialization strategy consists of the sequential composition of

the two procedures *UDF* and *RU*, which we will describe in detail in Sections 5.2.1 and 5.2.2, respectively.

The Specialization Strategy

Input: The program $P_{\mathcal{K}}[\varphi]$.
Output: A specialized program P_{sp} such that:

$$prop \in M(P_{\mathcal{K}}[\varphi]) \quad \text{iff} \quad prop \in M(P_{sp})$$

$UDF(P_{\mathcal{K}}[\varphi], P_A)$;
$RU(P_A, P_{sp})$

5.2.1. The UDF procedure

As briefly described in Section 5.1, the *UDF* procedure consists in the iterated application of the unfolding, definition, folding rules, possibly interleaved with applications of the clause removal and constraint replacement rules. The unfolding rule is applied according to the *Unfold* auxiliary procedure, and the definition and folding rules are applied according to the *Generalize&Fold* auxiliary procedure. These auxiliary procedures are described below.

In order to control the generalizations to be performed, we follow an approach which is similar to one considered in the context of partial deduction [46, 37]. We consider the set *Defs* made out of the clauses introduced by the definition rule during the *UDF* procedure together with clause γ_2: $negprop \leftarrow sat(X, init \wedge \neg\varphi)$, and we arrange this set *Defs* as a tree whose root is clause γ_2. Although γ_2 is not introduced by the definition rule R1, by abuse of language we will refer to *Defs* as a *set of definitions* or as a *tree of definitions*. Since, by construction, all clauses in *Defs* are distinct, we will identify each clause with a node of that tree. During the construction of *Defs* the *UDF* procedure may mark some of its nodes as '*terminated*', which means that they have no son-clauses.

Initially, *Defs* consists of clause γ_2 only, which is not marked as 'terminated'. The *UDF* procedure considers a leaf γ of *Defs* which is not marked as 'terminated' and unfolds it, by using the *Unfold* procedure, thereby deriving a set Γ of clauses. Then the *Generalize&Fold* procedure introduces a set *NewDefs* of new definitions such that, for each constrained literal of the form $c_1(X) \wedge sat(X, \psi_1)$ or $c_1(X) \wedge \neg sat(X, \psi_1)$ occurring in the body of a clause in Γ, there exists in $Defs \cup NewDefs$ a definition of the form $newp(X) \leftarrow d(X) \wedge sat(X, \psi_1)$ such that $\mathcal{D} \models \forall X\,(c_1(X) \rightarrow d(X))$. Thus, by folding the clauses in Γ w.r.t. all *sat* literals, we derive a new set Φ of clauses without occurrences of *sat* literals. The clauses of *NewDefs* are added to *Defs* as son-clauses of γ. If *NewDefs* is empty, then γ has no son-clauses and is marked as 'terminated'. When all leaf-clauses of *Defs* are marked as 'terminated', the *UDF* procedure halts, and the output program is obtained by collecting all clauses defining the predicates on which the predicate *prop* depends (in particular, *prop* does not depend on *sat* in the program derived by the *UDF* procedure and, thus, the clauses for *sat* are dropped).

In the *UDF* procedure we will use the functions *leaf* and *add-son-clauses* defined as follows. Given a clause γ, $leaf(\gamma)$ is a tree with precisely one node consisting of clause γ. Given a tree *Defs* of clauses, a leaf-clause γ, and a set *NewDefs* of clauses, $add\text{-}son\text{-}clauses(Defs, \gamma, NewDefs)$ is the tree of clauses obtained from *Defs* by adding a son-clause δ of γ for each clause δ occurring in *NewDefs*.

Procedure $UDF(P_{\mathcal{K}}[\varphi], P_A)$

Input: The program $P_{\mathcal{K}}[\varphi]$.
Output: A program P_A such that:

$prop \in M(P_{\mathcal{K}}[\varphi])$ iff $prop \in M(P_A)$.

$P_T := P_{\mathcal{K}}[\varphi]$; $Defs := leaf(\gamma_2)$, where γ_2 is $negprop \leftarrow sat(X, init \wedge \neg\varphi)$;
WHILE there exists a clause $\gamma \in P_T$ which is a leaf of $Defs$ and is not marked as 'terminated'
DO $Unfold(\gamma, \Gamma)$;
$Generalize\&Fold(Defs, \gamma, \Gamma, NewDefs, \Phi)$;
$P_T := (P_T - \{\gamma\}) \cup NewDefs \cup \Phi$;
$Defs := add\text{-}son\text{-}clauses(Defs, \gamma, NewDefs)$;
IF $NewDefs = \emptyset$ THEN mark γ as 'terminated'
END-WHILE;
$P_A := Def^*(prop, P_T)$

The Unfold Procedure.
The *Unfold* procedure takes as input a clause $\gamma \in P_T$ of the form $H \leftarrow c(X) \wedge sat(X, \psi)$, and returns as output a set of clauses derived from γ by applying the positive unfolding rule R2, the clause removal rule R6, and the constraint replacement rule R7. The *Unfold* procedure first unfolds γ w.r.t. $sat(X, \psi)$, and then applies the positive unfolding rule zero or more times as long as in the bodies of the clauses derived from δ there are atoms of one of the following forms: (i) $t(s_1, s_2)$, (ii) $ts(s, ss)$, (iii) $sat(s, e)$, where e is an elementary property, (iv) $sat(s, \neg\psi_1)$, (v) $sat(s, \psi_1 \wedge \psi_2)$, (vi) $sat(s, ex(\psi_1))$, and (vii) $sat\_all(ss, \psi_1)$.

Due to the structure of the clauses defining the predicates t, ts, sat, and sat_all, the *Unfold* procedure terminates for any ground CTL formula ψ occurring in γ (see Lemma 5.1 below). Note that, in particular, a clause is not unfolded w.r.t. any atom of the form $sat(X, af(\psi_1))$ or $sat(X, eu(\psi_1))$, because in these cases the recursive structure of the clauses defining sat may cause nontermination.

Then the set of clauses derived from γ by applying the unfolding rule is simplified by: (i) using rule R6s, thereby removing all clauses which are subsumed by a constrained fact, and (ii) using rule R7, thereby applying the *solve* function to the constraints occurring in the bodies of the clauses. Rules R6s and R7 are applied according to the following two auxiliary procedures: *Remove-Subsumed* and *Solve-Constraints*. Given a set Γ_1 of clauses in program P_T, (i) $Remove\text{-}Subsumed(\Gamma_1, \Gamma_2)$ derives a set Γ_2 of clauses by removing from Γ_1 every clause δ subsumed by a constrained fact in P_T different from δ, and (ii) $Solve\text{-}Constraints(\Gamma_1, \Gamma_2)$ derives a set Γ_2 of clauses by applying the *solve* function to every constraint occurring in the body of a clause in Γ_1, that is, $\Gamma_2 = \{(H \leftarrow solve(c, Y) \wedge G) \mid (H \leftarrow c \wedge G) \in \Gamma_1 \text{ and } Y = FV(c) \cap (FV(H) \cup FV(G))\}$.

Procedure $Unfold(\gamma, \Gamma)$
Input: A clause γ of the form $H \leftarrow c(X) \wedge sat(X, \psi)$ in program P_T.
Output: A set Γ of clauses.

$\Gamma_u := \{\gamma_u \mid \gamma_u \text{ is derived by unfolding } \gamma \text{ w.r.t. } sat(X, \psi)\}$;
WHILE there exists a clause $\delta : K \leftarrow d \wedge G_1 \wedge A \wedge G_2$ in Γ_u,
where A is of one of the following forms:
(i) $t(s_1, s_2)$, (ii) $ts(s, ss)$, (iii) $sat(s, e)$, where e is an elementary property,

(iv) $sat(s, \neg\psi_1)$, (v) $sat(s, \psi_1 \wedge \psi_2)$, (vi) $sat(s, ex(\psi_1))$
DO $\Gamma_u := (\Gamma_u - \{\delta\}) \cup \{\delta_u \mid \delta_u$ is derived by unfolding δ w.r.t. $A\}$
END-WHILE ;
WHILE there exists a clause $\delta : K \leftarrow d \wedge G_1 \wedge sat\_all(ss, \psi_1) \wedge G_2$ in Γ_u
DO $\Gamma_u := (\Gamma_u - \{\delta\}) \cup \{\delta_u \mid \delta_u$ is derived by unfolding δ w.r.t. $sat\_all(ss, \psi_1)\}$
END-WHILE ;
Remove-Subsumed(Γ_u, Γ_s);
Solve-Constraints(Γ_s, Γ)

Examples of application of the *Unfold* procedure will be given in Section 5.3.

Note that, during an application of the *Unfold* procedure, only the instance R6s of rule R6 is used to remove clauses. The other instance of rule R6, that is, rule R6u, is used to remove useless clauses during the *RU* procedure, which is executed after the *UDF* procedure.

Lemma 5.1. *The Unfold procedure terminates.*

Proof: Let us consider an application of the *Unfold* procedure with input clause γ and input program P_T. By analyzing the *UDF* procedure it can be verified that the clauses of P_T defining the predicates t, ts, sat, and sat_all are the ones of $P_{\mathcal{K}}[\varphi]$.

Let us now show that the execution of the first WHILE-DO statement terminates. Since at each unfolding step we derive a finite number of clauses, it is enough to prove the finiteness of every sequence of clauses $\delta_1, \delta_2, \ldots$ such that δ_1 belongs to the initial set Γ_u of clauses (see the first statement of the *Unfold* procedure) and δ_{i+1} is derived by unfolding δ_i during the execution of the first WHILE-DO statement.

Let us consider the well-founded ordering $>_c$ over clauses defined as follows. We introduce a function μ from goals to natural numbers such that: (i) for every constraint c, we have $\mu(c) = 0$, (ii) for every atom of the form $t(s_1, s_2)$, we have $\mu(t(s_1, s_2)) = 1$, (iii) for every atom of the form $ts(s, ss)$, we have $\mu(ts(s, ss)) = 1$, (iv) for every atom of the form $sat(s, \psi)$, we have $\mu(sat(s, \psi)) = 1 + size(\psi)$, (v) for every other atom A, we have $\mu(A)) = 0$, and (vi) for every literal of the form $\neg A$, we have $\mu(\neg A) = \mu(A)$. With any constrained goal $c \wedge L_1 \wedge \ldots \wedge L_n$, we associate a *multiset* $ms(c \wedge L_1 \wedge \ldots \wedge L_n) = \{\mu(c), \mu(L_1), \ldots, \mu(L_n)\}$ (here we use $\{\ldots\}$ also to denote multisets). Now, given any two clauses δ_i and δ_j, we define $\delta_i >_c \delta_j$ iff $ms(bd(\delta_i)) \gg ms(bd(\delta_j))$, where $\gg$ is the *multiset ordering* over natural numbers [20]. Since $\gg$ is a well-founded ordering, we have that also $>_c$ is a well-founded ordering.

During the execution of the first WHILE-DO statement a clause δ_{i+1} is derived by unfolding δ_i w.r.t. an atom of one of the following forms: (i) $t(s_1, s_2)$, (ii) $ts(s, ss)$, (iii) $sat(s, e)$, where e is an elementary property, (iv) $sat(s, \neg\psi_1)$, (v) $sat(s, \psi_1 \wedge \psi_2)$, and (vi) $sat(s, ex(\psi_1))$. By looking at the clauses defining the predicates t, ts, and sat (see Definition 8), the reader can verify that $\delta_i >_c \delta_{i+1}$. This proves that the sequence $\delta_1, \delta_2, \ldots$ is finite.

Next we show that the execution of the second WHILE-DO statement terminates. Note that the first WHILE-DO statement performs unfolding steps w.r.t. all atoms of the form $ts(s, ss)$. Thus, by the definition of ts (see Definition 7), upon termination of the execution of the first WHILE-DO statement every atom of the form $sat\_all(ss, \psi_1)$ occurring in the body of a clause in Γ_u will have ss bound to a list of the form $[s_1, \ldots, s_k]$, with $k \geq 1$, where $s_1, \ldots, s_k$ are terms representing states. Thus, the termination of the execution of the second WHILE-DO statement easily follows from the definition of the predicate sat_all (see Definition 8).

Finally, since we perform at most one application of the clause removal rule and constraint replacement rule for each clause derived by unfolding, we have that the *Unfold* procedure terminates. □

The Generalize&Fold Procedure
The choice of a suitable generalization technique is a difficult task because it should meet two somewhat conflicting requirements. On the one hand, generalization should ensure the termination of the specialization strategy by enforcing that the set of new definitions introduced during the application of the *UDF* procedure is finite. On the other hand, generalization should ensure that as many properties as possible can be proved. This issue has already been illustrated through Examples 4 and 5 in Section 5.1, where we have seen that: (i) if we make no generalization then we get nontermination, (ii) a suitable generalization (the one from $X=1$ to $X\geq 1$) allows us to prove the property of interest, and (iii) overgeneralization (the one from $X=1$ to *true*) prevents us to prove the property of interest.

The *Generalize&Fold* procedure proposed in this paper ensures the termination of the specialization strategy by extending to constraint logic programs some techniques for controlling generalization during *positive supercompilation* [65] and *partial deduction* [39, 36]. Our technique is based on the combined use of a *well-quasi-ordering* relation [20] and of a *generalization* operator defined on the set of clauses introduced by the definition rule during the *UDF* procedure. The use of a well-quasi-ordering guarantees that generalization is eventually applied and the properties of our clause generalization operator guarantee that each definition can be generalized a finite number of times only. The combined use of a well-quasi-ordering and a generalization operator ensures that a finite number of definitions is introduced during the *UDF* procedure.

Definition 11 (Well-Quasi-Ordering) *A* well-quasi-ordering *(wqo, for short) on a set S is a reflexive, transitive, binary relation $\preceq$ such that, for every infinite sequence $e_1, e_2, \ldots$ of elements of S, there exist i and j such that $i < j$ and $e_i \preceq e_j$.*

Given $e_1, e_2 \in S$ we write $e_1 \prec e_2$ if $e_1 \preceq e_2$ and not $e_2 \preceq e_1$, and we write $e_1 \approx e_2$ if $e_1 \preceq e_2$ and $e_2 \preceq e_1$. We say that the wqo $\preceq$ is thin if for all $e \in S$ the set $\{e' \in S \mid e \approx e'\}$ is finite.

The following property of wqo's, whose proof is left to the reader, will be used in our termination proof for the *UDF* procedure.

Proposition 5.2. *Suppose that $\preceq$ is a thin wqo on a set S. Then, for every infinite sequence $e_1, e_2, \ldots$ of distinct elements of S, there exist i and j such that $i < j$ and $e_i \prec e_j$.*

We will now show how a wqo on the atomic constraints of $\mathcal{C}$ can be extended to a wqo on $\mathcal{C}$. For reasons of simplicity, we will assume that every $c \in \mathcal{C}$ is equivalent to a conjunction $c_1 \wedge \ldots \wedge c_m$ of atomic constraints, that is, we will assume that $\mathcal{C}$ is *closed under projection*, as specified by the following definition.

Definition 12 (Closure Under Projection) *A set $\mathcal{C}$ of constraints is* closed under projection *if for every $c \in \mathcal{C}$ there exist m atomic constraints $c_1 \in \mathcal{C}, \ldots, c_m \in \mathcal{C}$, such that $\mathcal{D} \models \forall (c \leftrightarrow (c_1 \wedge \ldots \wedge c_m))$, where $\mathcal{D}$ is the given constraint interpretation.*

Thus, if $\mathcal{C}$ is closed under projection, then all existential quantifiers can be eliminated from constraints in $\mathcal{C}$.

Now, let us assume that $\mathcal{C}$ is closed under projection. A wqo $\preceq$ on the atomic constraints of $\mathcal{C}$ can be extended to a wqo on $\mathcal{C}$, still denoted by $\preceq$, as follows:

$c_1 \wedge \ldots \wedge c_m \preceq d_1 \wedge \ldots \wedge d_n$ if, for $i = 1, \ldots, m$, there exists j, with $1 \leq j \leq n$, such that $c_i \preceq d_j$.

We have the following property whose proof is left to the reader.

Proposition 5.3. *$\preceq$ is a thin wqo on $\mathcal{C}$ iff $\preceq$ is a thin wqo on the atomic constraints of $\mathcal{C}$.*

For our specialization examples of Sections 5.3 and 6 we will use the wqo defined in the following example.

Example 6. [Wqo on Linear Constraints] Let us consider the set Lin_k of constraints, whose atomic constraints are inequations (constructed by using the predicates $<$ and $\leq$) with k variables $X_1, \ldots, X_k$ and integer coefficients. Recall that an equation is considered as an abbreviation for the conjunction of two inequations. Let $\mathcal{Q}_{Lin}$ be the constraint interpretation for Lin_k introduced in Example 1. Lin_k is closed under projection and, thus, without loss of generality, we may assume that any constraint $c \in Lin_k$ is of the form $c_1 \wedge \ldots \wedge c_m$, where, for $i = 1, \ldots, m$: (1) c_i is an atomic constraint of the form $p_i \leq 0$ or $p_i < 0$, (2) p_i is a polynomial of the form $a^i_0 + a^i_1 X_1 + \ldots a^i_k X_k$, and (3) $a^i_0, a^i_1, \ldots, a^i_k$ are integer coefficients. For any atomic constraint c_i of the form specified above, we define $maxcoeff(c_i) = max\{|a^i_0|, |a^i_1|, \ldots, |a^i_k|\}$, and for any two atomic constraints b_1, b_2, we define:

$b_1 \preceq_{mc} b_2$ if $maxcoeff(b_1) \leq maxcoeff(b_2)$

The relation $\preceq_{mc}$ is a thin wqo on the atomic constraints of Lin_k and it can be extended to a thin wqo on Lin_k, also denoted $\preceq_{mc}$. □

Now, we extend the notion of wqo between constraints to the notion of wqo between the definitions introduced by the *Generalize&Fold* procedure. Every definition introduced by the *Generalize&Fold* procedure is a clause of the form $newp(X) \leftarrow c(X) \wedge sat(X, \psi)$ and for this reason we now introduce the notions of wqo and generalization operator for clauses of this form.

Definition 13. *Let $\preceq$ be a wqo on the set $\mathcal{C}$ of constraints. Given two clauses δ_1 and δ_2 of the form:*

$\delta_1 : newp_1(X) \leftarrow c_1(X) \wedge sat(X, \psi_1)$
$\delta_2 : newp_2(X) \leftarrow c_2(X) \wedge sat(X, \psi_2)$

we define $\delta_1 \preceq \delta_2$ if $c_1(X) \preceq c_2(X)$ and $\psi_1 = \psi_2$. We write $\delta_1 \prec \delta_2$ if $\delta_1 \preceq \delta_2$ and not $\delta_2 \preceq \delta_1$.

For any CTL formula φ, let Δ_φ be the set of clauses of the form $newp(X) \leftarrow c(X) \wedge sat(X, \psi)$, where ψ is a subformula of φ (including φ itself). We say that two clauses in Δ_φ are *equivalent modulo renaming* if one clause can be obtained from the other by renaming the head predicate symbol and the variables.

The following lemma, which will be used in the proof of termination of the *UDF* procedure, is a straightforward consequence of the fact that the set of the subformulas of φ is finite.

Lemma 5.4. *Let D be a subset of Δ_φ such that no two clauses in D are equivalent modulo renaming. For any given CTL formula φ, the relation $\preceq$ is a wqo on D iff $\preceq$ is a wqo on the set $\mathcal{C}$ of all constraints. Moreover, $\preceq$ is a thin wqo on D iff $\preceq$ is a thin wqo on $\mathcal{C}$.*

In order to introduce our generalization operator, we define a partial order $\sqsubseteq$ on the set $\mathcal{C}$ of all constraints with constraint interpretation $\mathcal{D}$, as follows: for any two constraints c_1 and c_2 in $\mathcal{C}$, we say that c_2 *is more general than* (or *is a generalization of*) c_1, and we write $c_1 \sqsubseteq c_2$ if $\mathcal{D} \models \forall\, (c_1 \rightarrow c_2)$. Given two clauses δ_1 and δ_2 of the form:

$\delta_1 : newp_1(X) \leftarrow c_1(X) \wedge sat(X, \psi)$

$\delta_2 : newp_2(X) \leftarrow c_2(X) \wedge sat(X, \psi)$

we say that δ_2 *is more general than* (or *is a generalization of*) δ_1, and we write $\delta_1 \sqsubseteq \delta_2$ if $c_1(X) \sqsubseteq c_2(X)$.

The following lemma follows directly from the definition of the folding rules R4 and R5, and from the definition of $\sqsubseteq$.

Lemma 5.5. *If a clause γ can be folded using a clause δ (by applying either rule R4 or Rule R5), and ϑ is a clause such that $\delta \sqsubseteq \vartheta$, then γ can also be folded using clause ϑ.*

Definition 14 (Generalization Operator) *Let $\precsim$ be a wqo on $\mathcal{C}$. A* generalization *operator on $\mathcal{C}$ is a binary operator $\ominus$ such that, for all constraints c, d in $\mathcal{C}$, we have:*

(1) $d \sqsubseteq c \ominus d$, and

(2) $c \ominus d \precsim c$.

Note that, in general, $\ominus$ is not commutative.

Our generalization operator $\ominus$ bears some similarities with the *widening* operator ∇ used in the field of abstract interpretation [13, 14]. In particular, similarly to the case of widening, every infinite sequence constructed by using the generalization operator stabilizes, that is, for every infinite sequence $d_1, d_2, \ldots$, if we consider the infinite sequence $c_1, c_2, \ldots$ of constraints defined as follows:

$c_1 = d_1$

$c_{i+1} = c_i \ominus d_{i+1}$, for $i \geq 1$,

then there exists m such that, for every $n > m \geq 1$, $c_m = c_n$.

However, there are also some differences between our generalization operator and the widening operator. Indeed, $c \nabla d$ is required to be an upper bound of both c and d (w.r.t. $\sqsubseteq$), while $c \ominus d$ is required to be an upper bound (w.r.t. $\sqsubseteq$) of d only. Moreover, for our generalization operator we have required that $c \ominus d \precsim c$, while for the widening operator no similar property is required.

Let us consider again the case where $\mathcal{C}$ is closed under projection (as it is the case for Lin_k). Let $\precsim$ be a thin wqo on $\mathcal{C}$ and let c, d be two constraints in $\mathcal{C}$ which are conjunctions of atomic constraints of the forms $c_1 \wedge \ldots \wedge c_m$ and $d_1 \wedge \ldots \wedge d_n$, respectively. We define:

$c \ominus d = c_{i1} \wedge \ldots \wedge c_{ir} \wedge d_{j1} \wedge \ldots \wedge d_{js}$

where: (1) $\{c_{i1}, \ldots, c_{ir}\} = \{c_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq c_h\}$, and
(2) $\{d_{j1}, \ldots, d_{js}\} = \{d_k \mid 1 \leq k \leq n \text{ and } d_k \precsim c\}$.

We leave it to the reader to show that, for any thin wqo $\precsim$, the operator $\ominus$ is indeed a generalization operator on $\mathcal{C}$, that is, $\ominus$ satisfies Conditions (1) and (2) of Definition 14. We say that $\ominus$ is the *generalization operator associated with* $\precsim$.

Example 7. Let us consider the wqo $\precsim_{mc}$ of Example 6 and the generalization operator $\ominus_{mc}$ associated with $\precsim_{mc}$. We have that $(X \geq 0 \wedge X < 1) \ominus_{mc} (X \geq 1 \wedge X \leq 2) = (X \geq 0 \wedge X \geq 1)$, which is equivalent to $X \geq 1$. Indeed, (i) $(X \geq 1 \wedge X \leq 2) \sqsubseteq (X \geq 0)$, (ii) $(X \geq 1 \wedge X \leq 2) \not\sqsubseteq (X < 1)$, (iii) $(X \geq 1) \precsim_{mc} (X \geq 0 \wedge X < 1)$, and (iv) it is not the case that $(X \leq 2) \precsim_{mc} (X \geq 0 \wedge X < 1)$. □

Similarly to what we have done for wqo's, now we extend any given generalization operator acting on pairs of constraints to a generalization operator acting on pairs of definitions introduced by the *Generalize&Fold* procedure.

Definition 15. *Let $\ominus$ be a generalization operator on $\mathcal{C}$. Given two clauses δ_1 and δ_2 of the form:*

δ_1*:* $newp_1(X) \leftarrow c_1(X) \wedge sat(X, \psi)$
δ_2*:* $newp_2(X) \leftarrow c_2(X) \wedge sat(X, \psi)$

the generalization of δ_1 and δ_2, denoted $\delta_1 \ominus \delta_2$, is the clause

$$genp(X) \leftarrow (c_1(X) \ominus c_2(X)) \wedge sat(X, \psi)$$

where genp is a new predicate symbol.

The following lemma follows directly from the definitions.

Lemma 5.6. *Let φ be a CTL formula. The operator $\ominus$ is a generalization operator on Δ_φ, that is, for any two clauses δ_1 and δ_2 in Δ_φ, we have that: (1) $\delta_2 \sqsubseteq \delta_1 \ominus \delta_2$, and (2) $\delta_1 \ominus \delta_2 \precsim \delta_1$.*

Thus, by Lemmata 5.5 and 5.6 every clause that can be folded using a clause δ_2 can also be folded using the generalization $\delta_1 \ominus \delta_2$ for any given clause δ_1.

We now present the *Generalize&Fold* procedure, which takes as input: (i) the tree *Defs* of definitions introduced during the *UDF* procedure by applying Rule R1, (ii) a clause γ which occurs at a leaf of *Defs*, (iii) a finite set Γ of clauses obtained by unfolding γ using the *Unfold* procedure, and (iv) a thin wqo $\precsim$ on constraints. The *Generalize&Fold* procedure returns as output: (i) a finite set *NewDefs* of new definitions, and (ii) a set Φ of clauses derived by folding the clauses in Γ w.r.t. all *sat* literals occurring in their bodies by using definitions taken either from *Defs* or from *NewDefs*. Thus, no clause in Φ contains occurrences of *sat* literals.

A clause $\delta \in \Gamma$ is folded w.r.t. a *sat* literal occurring in its body as follows. Suppose that δ is of the form $H \leftarrow d \wedge G_1 \wedge L \wedge G_2$, where L is either a literal of the form $sat(X, \psi)$ or a literal of the form $\neg sat(X, \psi)$. We consider a clause ζ of the form $newp(X) \leftarrow solve(d, \{X\}) \wedge sat(X, \psi)$, where *newp* is a new predicate symbol. Clause ζ is called a *folder* for δ. Clause δ can be folded w.r.t. $sat(X, \psi)$ or w.r.t. $\neg sat(X, \psi)$ using clause ζ. However, if at every application of the *Generalize&Fold* procedure we introduce all folder clauses for the clauses in Γ, then the *UDF* procedure may not terminate. In order to guarantee the termination of this procedure, we introduce suitable generalizations of folder clauses and we fold the clauses of Γ using these generalizations (indeed, by Lemma 5.5, a clause δ can be folded using any generalization of a folder clause for δ).

We consider the following two cases.
(1) If a generalization η of ζ exists in *Defs*, then we add no clause to *NewDefs* and we fold δ using η.
(2) Otherwise, if no generalization of ζ exists in *Defs*, we construct a generalization of ζ by matching this clause against γ and the ancestors of γ in *Defs*. We consider the path $\alpha_1, \ldots, \alpha_m$ of *Defs*, denoted by $anc(\gamma, Defs)$, where α_1 is the root of *Defs* (that is, α_1 is γ_2), and α_m is γ.
(2.1) If there exists a clause α in $anc(\gamma, Defs)$ such that $\alpha \prec \zeta$ and β is the rightmost (that is, last generated) such clause, then we apply the generalization operator and we introduce the clause $\vartheta = \beta \ominus \zeta$. (Recall that the order $\prec$ is extended from constraints to clauses as indicated in Definition 13.) Then we add ϑ to *NewDefs* and we fold δ using ϑ.
(2.2) Otherwise, we add ζ to *NewDefs* and we fold δ using ζ.

In the *Generalize&Fold* procedure we use the following notation. Given a clause δ, by $folders(\delta)$ we denote the set of clauses which are folders for δ. By $fold(\delta, \vartheta)$ we denote a clause derived by folding clause δ using clause ϑ.

Procedure *Generalize&Fold*$(Defs, \gamma, \Gamma, NewDefs, \Phi)$
Input: (i) a tree *Defs* of definitions, (ii) a clause γ which is a non-terminated leaf of *Defs*, (iii) a set Γ of clauses obtained from γ by the *Unfold* procedure, and (iv) a thin wqo $\precsim$ on constraints.
Output: (i) A set *NewDefs* of new definitions, and (ii) a set Φ of folded clauses.

$NewDefs := \emptyset$; $\Phi := \Gamma$;
WHILE there exist a clause $\delta \in \Phi$ and a clause $\zeta \in folders(\delta)$ DO

***Generalize*:**
IF there exists a clause η in *Defs* such that $\zeta \sqsubseteq \eta$
THEN $\vartheta := \eta$
ELSE (IF there exists a clause α in $anc(\gamma, Defs)$ such that $\alpha \prec \zeta$ and
β is the rightmost clause in $anc(\gamma, Defs)$ such that $\beta \prec \zeta$
THEN $\vartheta := \beta \ominus \zeta$
ELSE $\vartheta := \zeta$;
$NewDefs := NewDefs \cup \{\vartheta\}$) ;

***Fold*:**
$\Phi := (\Phi - \{\delta\}) \cup \{fold(\delta, \vartheta)\}$;
END-WHILE

Examples of application of the *Generalize&Fold* procedure can be found in Section 5.3.

Lemma 5.7. *The Generalize&Fold procedure terminates.*

Proof: Each execution of the body of the WHILE-DO statement deletes from Φ one occurrence of *sat*. □

Now we show that the *UDF* procedure indeed preserves the perfect model semantics.

Theorem 5.8 (Correctness of the *UDF* Procedure) *Let $P_{\mathcal{K}}[\varphi]$ and P_A be the input and output programs, respectively, of the UDF procedure. Then*

$$prop \in M(P_{\mathcal{K}}[\varphi]) \quad \textit{iff} \quad prop \in M(P_A)$$

Proof: The use of the transformation rules according to the *UDF* procedure generates a transformation sequence $P_0, \ldots, P_n$ (see Section 4), where: (i) P_0 is $P_{\mathcal{K}}[\varphi]$ and (ii) P_n is the final value of program P_T. We will show that this transformation sequence satisfies the hypothesis of Theorem 4.1 presented at the end of Section 4. Let us consider an application of the positive folding rule R4, performed during an application of the *Generalize&Fold* procedure. Suppose that this application of R4 consists in folding clause δ using clause ϑ (see the statement $\Phi := (\Phi - \{\delta\}) \cup \{fold(\delta, \vartheta)\}$). Either ϑ occurs in *Defs* (that is, it has been introduced by the definition rule in a previous application of the *Generalize&Fold* procedure) or ϑ is added to *NewDefs* and, after the completion of the *Generalize&Fold* procedure, it is added to *Defs* (by the statement $Defs := add\text{-}son\text{-}clauses(Defs, \gamma, NewDefs)$ of the *UDF* procedure). Every clause in *Defs* is unfolded by using the *Unfold* procedure. Thus, the hypothesis of Theorem 4.1 is satisfied and, since *prop* occurs in $P_{\mathcal{K}}[\varphi]$, we have that $prop \in M(P_{\mathcal{K}}[\varphi])$ iff $prop \in M(P_T)$. Now,

by a property of perfect models, we have that $prop \in M(P_T)$ iff $prop \in M(Def^*(prop, P_T))$. Thus, the thesis follows from the fact that $P_A = Def^*(prop, P_T)$. □

We end this section by showing that the *UDF* procedure terminates.

Theorem 5.9. *The UDF procedure terminates for every input program* $P_{\mathcal{K}}[\varphi]$.

Proof: By Lemmata 5.1 and 5.7 each execution of the *Unfold* and *Generalize&Fold* procedures terminates. Hence, every execution of the body of the WHILE-DO statement terminates. At each execution of the body of the WHILE-DO statement the *UDF* procedure adds zero or more new clauses to the tree *Defs* of definitions. Let $Defs_\lambda$ be the *limit value* of *Defs*, that is, $Defs_\lambda$ is the final value of $Defs_\lambda$, if the *UDF* procedure terminates, and $Defs_\lambda$ is the infinite tree constructed by the perpetual execution of the *UDF* procedure, otherwise. Thus, the *UDF* procedure terminates if and only if $Defs_\lambda$ is a finite tree. Let us consider a maximally long, possibly infinite, path $\delta_1, \delta_2, \ldots$ of $Defs_\lambda$. We have that: (i) δ_1 is the input clause γ_2: $negprop \leftarrow sat(X, init \wedge \neg\varphi)$, and (ii) for $n \geq 1$, there exist values of Γ, *Defs*, *NewDefs*, and Φ such that $\delta_n \in Defs$ and $\delta_n{+}1$ is a member of the set *NewDefs* obtained from δ_n by applying the procedure $Unfold(\delta_n, \Gamma)$ followed by the procedure $Generalize\&Fold(Defs, \delta_n, \Gamma, NewDefs, \Phi)$.

Now we prove that the path $\delta_1, \delta_2, \ldots$ is finite. First, note that the following property holds:

Property (A) (i) $\delta_2, \delta_3, \ldots$ are clauses of the form $newp(X) \leftarrow c(X) \wedge sat(X, \psi)$, where ψ is a subformula of the CTL formula φ occurring in the input clause γ_2, and (ii) no two clauses in $\delta_2, \delta_3, \ldots$ are equivalent modulo renaming.

Point(i) follows from the fact that, by the structure of the clauses defining the *sat* predicate (see Definition 8), every *sat* literal generated during the *UDF* procedure is of the form $sat(X, \psi)$, where ψ is a subformula of φ. Point (ii) follows from the fact that the *Generalize&Fold* procedure never adds a new definition ζ to *NewDefs*, if in *Defs* there exists a definition η such that $\zeta \sqsubseteq \eta$ and, if ζ is equivalent to η modulo renaming, then $\zeta \sqsubseteq \eta$.

By Property (A) and Lemma 5.4, $\precsim$ is a thin wqo on the set $\{\delta_2, \delta_3, \ldots\}$ and, thus, $\precsim$ is a thin wqo on $\{\delta_1, \delta_2, \ldots\}$. Next we show that the following property holds for the sequence $\delta_1, \delta_2, \ldots$:

Property (B) for any i, j such that $i < j$, we have that $\delta_i \not\prec \delta_j$.

The proof of Property (B) proceeds by induction. We assume that this property holds for an initial segment of $\delta_1, \delta_2, \ldots$ say $\delta_1, \ldots, \delta_n$, and we show that it holds also for $\delta_1, \ldots, \delta_n, \delta_{n+1}$.

Clause δ_{n+1} is added to *NewDefs* by an application of the *Generalize&Fold* procedure with input clause δ_n and input tree *Defs*. Thus, $anc(\delta_n, Defs)$ is the sequence $\delta_1, \ldots, \delta_n$ and, according to the *Generalize* phase of the *Generalize&Fold* procedure, clause δ_{n+1} is computed as follows, for a suitable clause ζ:

IF there exists a clause α in $\delta_1, \ldots, \delta_n$ such that $\alpha \prec \zeta$ and
β is the rightmost clause in $\delta_1, \ldots, \delta_n$ such that $\beta \prec \zeta$
THEN $\delta_{n+1} := \beta \ominus \zeta$
ELSE $\delta_{n+1} := \zeta$

Thus, there are the following two cases:
(Case 1) There exists k, with $1 \leq k \leq n$, such that $\delta_k \prec \zeta$ and, for $j = k+1, \ldots, n$, $\delta_j \not\prec \zeta$. We have that $\delta_{n+1} = \delta_k \ominus \zeta$. Now we prove by contradiction that, for $i = 1, \ldots, n$, $\delta_i \not\prec \delta_{n+1}$. Assume to the contrary that, for some i, with $1 \leq i \leq n$, $\delta_i \prec \delta_{n+1}$. From Condition (2) of Definition 14, it follows that $\delta_{n+1} \precsim \delta_k$ and, therefore, we get $\delta_i \prec \delta_k$ and $\delta_i \prec \zeta$. If $i < k$ then we contradict the assumption that Property (B) holds for the sequence $\delta_1, \ldots, \delta_n$. If $i = k$ then

we get $\delta_i \prec \delta_i$, contradicting the irreflexivity of $\prec$. If $i > k$ then we contradict the fact that for $j = k+1, \ldots, n$, $\delta_j \not\prec \zeta$.
(Case 2) There is no clause α in $\delta_1, \ldots, \delta_n$ such that $\alpha \prec \zeta$. In this case $\delta_{n+1} = \zeta$.

In both cases, there is no clause δ_i in $\delta_1, \ldots, \delta_n$ such that $\delta_i \prec \delta_{n+1}$, and, thus, the sequence $\delta_1, \ldots, \delta_n, \delta_{n+1}$ enjoys Property (B). Now the finiteness of the sequence $\delta_1, \delta_2, \ldots$ follows from Property (B), from the fact that $\prec$ is a thin wqo, and from Proposition 5.2. Thus, $Defs_\lambda$ is a finite tree and we may conclude that the *UDF* procedure terminates. □

5.2.2. The RU procedure

The goal of the *RU* procedure is to derive decided predicates. In particular, this procedure has the objective of obtaining a final program P_{sp} where the predicate *prop* which encodes the property of interest is decided, that is, either P_{sp} contains the fact $prop \leftarrow$ or P_{sp} contains no clauses for *prop*.

Let us first prove that the program P_A which has been derived from $P_{\mathcal{K}}[\varphi]$ at the end of the *UDF* procedure, is stratified, that is, P_A has a finite number of strata.

Lemma 5.10. *Let P_A be the output program of the UDF procedure. Then P_A is stratified.*

Proof: Each predicate occurring in P_A is: either (i) *prop*, or (ii) *negprop*, or (iii) a predicate *newp* introduced during the *UDF* procedure. (In particular, no clause for *sat* which belongs to the encoding program $P_{\mathcal{K}}$ occurs in P_A.) Program P_A is stratified w.r.t. the level mapping λ defined as follows: (i) $\lambda(prop) = \lambda(negprop) + 1$, (ii) $\lambda(negprop) = \max\{\lambda(newp) \mid newp \text{ occurs in } P_A\}$, and (iii) $\lambda(newp) = size(\psi)$, where the definition of *newp* in *Defs* is $newp(X) \leftarrow c(X) \wedge sat(X, \psi)$. Indeed, by construction, for every clause γ in P_A of the form $newp(X) \leftarrow c(X) \wedge G$ and for all literals L in G we have that:
(1) if L is of the form $newq(Y)$ then $\lambda(newq) \leq \lambda(newp)$, and
(2) if L is of the form $\neg newq(Y)$ then $\lambda(newq) < \lambda(newp)$. □

By Lemma 5.10, the *RU* procedure may work bottom-up on the strata of P_A. This procedure simplifies the definition of every predicate p occurring in the program, with the aim of deriving either the fact $p \leftarrow$ or the empty definition for p. In order to do so the *RU* procedure makes use of two auxiliary procedures: (1) the *Remove-Clauses* procedure consisting of applications of the clause removal rule R6, and (2) the *Unfold-Decided* procedure consisting of applications of the positive and negative unfolding rules R2 and R3 w.r.t. decided literals.

Given a program P_{in} the procedure *Remove-Clauses*(P_{in}, P_{out}) derives a program P_{out} by applying the clause removal rule as follows: (i) a program P' is derived from P_{in} by applying rule R6u and deleting all clauses that are useless in P_{in}, and (ii) program P_{out} is derived from P' by applying rule R6s and deleting every clause γ that is subsumed by a constrained fact occurring in $P' - \{\gamma\}$.

Given a program P_{in} the procedure *Unfold-Decided*(P_{in}, P_{out}) derives a program P_{out} as follows: (i) a program P' is derived from P_{in} by applying the positive unfolding rule R2 and the negative unfolding rule R3 w.r.t. all decided literals occurring in bodies of clauses of P_{in}, and (ii) program P_{out} is derived from P' by applying the function $solve(c(X), \{X\})$ to every constraint occurring in the body of a clause in P'. (These applications of the function *solve* can be viewed as applications of the procedure *Solve-Constraints* introduced in Section 5.2.1 as a part of the procedure *Unfold*.)

Thus, the *Remove-Clauses* procedure may derive new decided predicates either: (i) by removing, via rule R6u, the whole definition of a useless predicate, or (ii) by removing from the definition of a predicate, via rule R6s, all clauses which are not constrained facts. Also the *Unfold-Decided* procedure may derive new decided predicates by unfolding the program clauses w.r.t. decided literals. For every stratum of the input program, the RU procedure iterates the execution of the *Remove-Clauses* procedure followed by the *Unfold-Decided* procedure so to derive new decided predicates, until a fixpoint is reached.

Procedure $RU(P_{in}, P_{out})$
Input: A stratified program P_{in}.
Output: A program P_{out} such that $M(P_{in}) = M(P_{out})$.

Let $S_1, \ldots, S_n$ be a stratification of program P_{in}.
$P_{out} := \emptyset$;
FOR $i := 1, \ldots, n$ DO
  $P_{out} := P_{out} \cup S_i$;
  REPEAT $P_1 := P_{out}$;
    Remove-Clauses(P_1, P_2);
    Unfold-Decided(P_2, P_3);
    $P_{out} := P_3$
  UNTIL $P_1 = P_3$
END-FOR

An example of application of the RU procedure will be given in Section 5.3.

The correctness of the RU procedure follows from the correctness of the transformation rules, as we now show.

Theorem 5.11 (Correctness of the *RU* Procedure) *Let P_{in} and P_{out} be the input and output programs, respectively, of the RU procedure. Then $M(P_{in}) = M(P_{out})$.*

Proof: The RU procedure constructs a transformation sequence $P_{in}, \ldots, P_{out}$ by using rules R2, R3, R6, and R7. The positive folding rule R4 is never applied and, therefore, the hypothesis of Theorem 4.1 is trivially satisfied. Thus, by Theorem 4.1, for every ground atom A whose predicate occurs in P_{in}, we have that $A \in M(P_{in})$ iff $A \in M(P_{out})$. The definition rule R1 is not applied when constructing the transformation sequence $P_{in}, \ldots, P_{out}$ and, hence, every predicate that occurs in P_{out} also occurs in P_{in}. Thus, for every ground atom A whose predicate occurs in $P_{in} \cup P_{out}$, we have that $A \in M(P_{in})$ iff $A \in M(P_{out})$. Hence, $M(P_{in}) = M(P_{out})$. □

The termination of the RU procedure is a consequence of the fact that by unfolding a clause γ w.r.t. a decided literal, γ is replaced by a set of clauses whose body has strictly fewer literals than the body of γ. Thus, there exists no infinite sequence of programs constructed by clause removal and by unfolding w.r.t. decided literals, and eventually, the exit condition $P_1 = P_3$ of the REPEAT-UNTIL statement of the RU procedure is true. Thus, we have the following result.

Theorem 5.12 (Termination of the *RU* Procedure) *Let P_{in} be a stratified program. Then the RU procedure terminates for the input program P_{in}.*

5.2.3. Soundness and Termination of the Verification Method

Now we use the results of Sections 5.2.1 and 5.2.2 to show that our verification method always terminates with a sound result. As already mentioned, no complete method exists.

As an immediate consequence of Theorems 5.8, 5.11, 5.9, and 5.12 we get the following two results, which establish the correctness and termination of our specialization strategy.

Theorem 5.13 (Correctness of the Specialization Strategy) *Let $P_{\mathcal{K}}[\varphi]$ and P_{sp} be the input and output programs, respectively, of the specialization strategy. Then $prop \in M(P_{\mathcal{K}}[\varphi])$ iff $prop \in M(P_{sp})$.*

Theorem 5.14 (Termination of the Specialization Strategy) *The specialization strategy terminates for every input program $P_{\mathcal{K}}[\varphi]$.*

Now, we can prove the soundness of our verification method based on program specialization.

Theorem 5.15 (Soundness of the Verification Method) *Let $\mathcal{K}$ be a Kripke structure and let φ be a CTL formula. Let P_{sp} be the output of the specialization strategy for the input program $P_{\mathcal{K}}[\varphi]$.*
If $Def(prop, P_{sp}) = \{prop \leftarrow\}$, then $\mathcal{K}, s \models \varphi$, for all initial states s of $\mathcal{K}$.
If $Def(prop, P_{sp}) = \emptyset$, then $\mathcal{K}, s \not\models \varphi$, for some initial state s of $\mathcal{K}$.

Proof: If $Def(prop, P_{sp}) = \{prop \leftarrow\}$ then $prop \in M(P_{sp})$ and, hence, by Theorem 5.13, $prop \in M(P_{\mathcal{K}}[\varphi])$. Thus, by Theorem 3.4, we have that $\mathcal{K}, s \models \varphi$, for all initial states s of $\mathcal{K}$.
If $Def(prop, P_{sp}) = \emptyset$ then $prop \notin M(P_{sp})$ and, hence, by Theorem 5.13, $prop \notin M(P_{\mathcal{K}}[\varphi])$. Thus, by Theorem 3.4, we have that $\mathcal{K}, s \not\models \varphi$, for some initial state s of $\mathcal{K}$. □

5.3. An Example of Application of the Specialization Strategy

Let us consider the Kripke structure $\mathcal{K}$ presented in Example 3 at the end of Section 3. We want to verify that, starting from the initial state $\langle a, 0\rangle$, there exists a computation path in $\mathcal{K}$ such that, for all states $\langle X_1, X_2\rangle$ along that path, we have that $X_2 \geq 0$. This property is expressed by the relation $\mathcal{K}, \langle a, 0\rangle \models \neg AF(neg)$ which asserts that the CTL formula $\neg AF(neg)$ is true in the initial state $\langle a, 0\rangle$ of $\mathcal{K}$.

In order to verify this property, we consider the program $P_{\mathcal{K}}[\neg AF(neg)]$, that is, $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, where: (i) $P_{\mathcal{K}}$ is the encoding program constructed in Example 3 according to Definition 8, and (i) γ_1 and γ_2 are the following clauses:

γ_1: $prop \leftarrow \neg negprop$
γ_2: $negprop \leftarrow sat(\langle X_1, X_2\rangle, init \wedge af(neg))$

We will prove that $\mathcal{K}, \langle a, 0\rangle \models \neg AF(neg)$ by applying the specialization strategy to $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ and deriving a program P_{sp} where the predicate *prop* is defined by the fact $prop \leftarrow$.

First we apply the *UDF* procedure. Initially, program P_T is $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ and *Defs* is a tree of clauses consisting of clause γ_2 only. The tree *Defs* is represented as the set of its root-to-leaf paths, each of which is of the form $(\delta_1, \ldots, \delta_n)$, where δ_1 is the root γ_2 of *Defs*, δ_n is a leaf-clause, and for $i = 1, \ldots, n-1$, with $n \geq 1$, clause δ_{i+1} is a son-clause of δ_i. A clause δ marked as 'terminated' is denoted by $\delta^{\dagger}$, and a terminated clause may occur only at the right end of a path. Since in the tree *Defs* every clause occurs only once, from a given set of paths there exists at most one tree *Defs* represented by that set.

First iteration.
We consider γ_2, which is a non-terminated leaf of *Defs*.

Unfold. By applying the unfolding rule according to the *Unfold* procedure, from clause γ_2 we derive

δ_1: $\mathit{negprop} \leftarrow X_1 = a \wedge X_2 = 0 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

Note that the *Unfold* procedure halts because the last argument of *sat* in the body of δ_1 is of the form $\mathit{af}(\ldots)$.

Generalize&Fold. For the application of the *Generalize&Fold* auxiliary procedure we will use the wqo defined as follows. The constraints used in $P_{\mathcal{K}}$ are of the form $X_1 = s \wedge c(X_2)$, where $s \in \{a, b\}$ and $c(X_2)$ is a conjunction of linear equations and inequations with integer coefficients (see Examples 1 and 3). As already mentioned, an equation of the form $p_1 = p_2$, where p_1 and p_2 are polynomials, is considered as an abbreviation for $p_1 - p_2 \leq 0 \wedge p_1 - p_2 \geq 0$.

We define $(X_1 = s_1 \wedge c_1(X_2)) \precsim_{emc} (X_1 = s_2 \wedge c_2(X_2))$ if s_1 is equal to s_2 and $c_1(X_2) \precsim_{mc} c_2(X_2)$, where $\precsim_{mc}$ is the wqo defined in Example 6. Since $\precsim_{mc}$ is a thin wqo and $\{a, b\}$ is a finite set, we have that also $\precsim_{emc}$ is a thin wqo. We will also consider the extension of $\precsim_{emc}$ to clauses (see Definition 13) and the associated generalization operator for clauses (see Definition 15), denoted $\ominus_{emc}$.

Let us now execute *Generalize&Fold* where the input is: (i) the tree *Defs* of clauses consisting of clause γ_2 only, (ii) clause γ_2, and (iii) the set $\Gamma = \{\delta_1\}$ of clauses derived from γ_2 by the *Unfold* procedure. We have that $\mathit{anc}(\gamma_2, \mathit{Defs})$ is the sequence consisting of clause γ_2 only. The only folder clause for δ_1 is:

δ_2: $\mathit{new1}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 0 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

There is no clause η in *Defs* such that $\delta_2 \sqsubseteq \eta$ and there is no clause α in $\mathit{anc}(\gamma_2, \mathit{Defs})$ such that $\alpha \prec_{emc} \delta_2$ (indeed, $\gamma_2 \not\prec_{emc} \delta_2$). Therefore, we fold δ_1 using δ_2, thereby deriving:

γ_3: $\mathit{negprop} \leftarrow X_1 = a \wedge X_2 = 0 \wedge \mathit{new1}(\langle X_1, X_2\rangle)$

We replace clause γ_2 in P_T by clauses γ_3 and δ_2, and we add δ_2 to *Defs* as a non-terminated son-clause of γ_2. Thus, $P_T = P_{\mathcal{K}} \cup \{\gamma_1, \gamma_3, \delta_2\}$ and $\mathit{Defs} = \{(\gamma_2, \delta_2)\}$.

Second iteration.
We consider $\delta_2 \in P_T$, which is a non-terminated leaf-clause of *Defs*.

Unfold. By unfolding and constraint replacement, from δ_2 we derive:

δ_3: $\mathit{new1}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_1 = a \wedge Y_2 = 2 \wedge \mathit{sat}(\langle Y_1, Y_2\rangle, \mathit{af}(\mathit{neg}))$

Generalize&Fold. The only folder clause for δ_3 is:

δ_4: $\mathit{new2}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 2 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

whose body has been obtained from the body of δ_3 by applying the *solve* function and then renaming variables. In *Defs* there is no clause η such that $\delta_4 \sqsubseteq \eta$. Indeed, $\delta_4 \not\sqsubseteq \gamma_2$ and $\delta_4 \not\sqsubseteq \delta_2$. However, δ_2 is a clause in $\mathit{anc}(\delta_2, \mathit{Defs})$ such that $\delta_2 \prec_{emc} \delta_4$ (because $\mathit{maxcoeff}(X_2 \leq 0) < \mathit{maxcoeff}(X_2 - 2 \leq 0)$ and $\mathit{maxcoeff}(X_2 \geq 0) < \mathit{maxcoeff}(X_2 - 2 \geq 0)$). Thus, we use the generalization operator $\ominus_{emc}$ and we introduce the following clause $\delta_5 =_{\mathit{def}} \delta_2 \ominus_{emc} \delta_4$:

δ_5: $\mathit{new3}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 \geq 0 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

(When computing the generalization, recall that a linear equation with integer coefficients is considered as an abbreviation for the conjunction of two linear inequations). Now, we fold δ_3 using δ_5 and we derive:

γ_4: $\mathit{new1}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_1 = a \wedge Y_2 = 2 \wedge \mathit{new3}(\langle Y_1, Y_2\rangle)$

We replace clause δ_2 in P_T by clauses γ_4 and δ_5, and we add δ_5 to *Defs* as a non-terminated son-clause of δ_2. Thus, $P_T = P_{\mathcal{K}} \cup \{\gamma_1, \gamma_3, \gamma_4, \delta_5\}$ and $\mathit{Defs} = \{(\gamma_2, \delta_2, \delta_5)\}$.

Third iteration.
We consider $\delta_5 \in P_T$, which is a non-terminated leaf-clause of *Defs*.

Unfold. By unfolding and constraint replacement, from δ_5 we derive:

δ_6: $\mathit{new3}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_1 = a \wedge Y_2 = 2 \wedge \mathit{sat}(\langle Y_1, Y_2\rangle, \mathit{af}(\mathit{neg}))$

δ_7: $\mathit{new3}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_1 = a \wedge Y_2 = X_2 + 2 \wedge Y_3 = b \wedge Y_4 = X_2 \wedge \mathit{sat}(\langle Y_1, Y_2\rangle, \mathit{af}(\mathit{neg})) \wedge \mathit{sat}(\langle Y_3, Y_4\rangle, \mathit{af}(\mathit{neg}))$

Generalize&Fold. Clause δ_6 has the following folder clause:

δ_8: $\mathit{new4}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 2 \wedge \mathit{sat}(X_1, X_2, \mathit{af}(\mathit{neg}))$

We have that $\delta_8 \sqsubseteq \delta_5$ and, therefore, δ_8 is not added to *Defs*. We fold δ_6 using δ_5 and we get:

γ_5: $\mathit{new3}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_1 = a \wedge Y_2 = 2 \wedge \mathit{new3}(\langle Y_1, Y_2\rangle)$

Clause δ_7 has the following two folder clauses:

δ_9: $\mathit{new5}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 > 2 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

δ_{10}: $\mathit{new6}(\langle X_1, X_2\rangle) \leftarrow X_1 = b \wedge X_2 > 0 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

We have that $\delta_9 \sqsubseteq \delta_5$. We also have that $\delta_{10} \not\sqsubseteq \eta$ for any clause η in *Defs*, because the constraint $X_1 = b$ does not occur in any clause in *Defs*. For the same reason, $\alpha \not\prec_{emc} \delta_{10}$ for any clause α in $\mathit{anc}(\delta_5, \mathit{Defs})$. Thus, we fold w.r.t. the two *sat* literals in δ_7 using clauses δ_5 and δ_{10}, and we derive:

γ_6: $\mathit{new3}(\langle X_1, X_2\rangle) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_1 = a \wedge Y_2 = X_2 + 2 \wedge Y_3 = b \wedge Y_4 = X_2 \wedge \mathit{new3}(\langle Y_1, Y_2\rangle) \wedge \mathit{new6}(\langle Y_3, Y_4\rangle)$

We replace clause δ_5 in P_T by clauses γ_5, γ_6, and δ_{10}, and we add δ_{10} to *Defs* as a non-terminated son-clause of δ_5. Thus, $P_T = P_{\mathcal{K}} \cup \{\gamma_1, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \delta_{10}\}$ and $\mathit{Defs} = \{(\gamma_2, \delta_2, \delta_5, \delta_{10})\}$.

Fourth iteration.
We consider $\delta_{10} \in P_T$, which is a non-terminated leaf-clause of *Defs*.

Unfold. By unfolding and constraint replacement, from δ_{10} we derive:

δ_{11}: $\mathit{new6}(\langle X_1, X_2\rangle) \leftarrow X_1 = b \wedge X_2 > 0 \wedge Y_1 = b \wedge Y_2 = X_2 + 1 \wedge \mathit{sat}(\langle Y_1, Y_2\rangle, \mathit{af}(\mathit{neg}))$

Generalize&Fold. Clause δ_{11} has the following folder clause:

δ_{12}: $\mathit{new6}(\langle X_1, X_2\rangle) \leftarrow X_1 = b \wedge X_2 > 1 \wedge \mathit{sat}(\langle X_1, X_2\rangle, \mathit{af}(\mathit{neg}))$

Since $\delta_{12} \sqsubseteq \delta_{10}$, we fold δ_{11} using δ_{10} and we get:

γ_7: $\mathit{new6}(\langle X_1, X_2\rangle) \leftarrow X_1 = b \wedge X_2 > 0 \wedge Y_1 = b \wedge Y_2 = X_2 + 1 \wedge \mathit{new6}(\langle Y_1, Y_2\rangle)$

We replace clause δ_{10} in P_T by clause γ_7 and, since no new definition clause has been introduced by the *Generalize&Fold* procedure, we mark δ_{10} as a terminated leaf-clause of *Defs*. Thus, $P_T = P_{\mathcal{K}} \cup \{\gamma_1, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7\}$ and $\mathit{Defs} = \{(\gamma_2, \delta_2, \delta_5, \delta_{10}^{\dagger})\}$.

The only leaf-clause δ_{10} of *Defs* is marked as terminated (as indicated by the † superscript) and, thus, we exit from the WHILE-DO loop. By deleting from P_T all clauses for *sat* on which *prop* does not depend, we get the output program of the *UDF* procedure, which is $P_A = \{\gamma_1, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7\}$.

Now we apply the *RU* procedure to the input program P_A and we derive the specialized program P_{sp} as follows. We compute a stratification of program P_A and we get $P_A = S_1 \cup S_2$, where $S_1 = \{\gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7\}$ and $S_2 = \{\gamma_1\}$. Then, we compute P_{sp} by processing the two strata of S_1 and S_2 as described below.

Stratum 1. Since the predicates *negprop*, *new*1, *new*3, and *new*6 are useless in stratum S_1, we remove their definitions and we derive $S_1' = \emptyset$.

Stratum 2. We consider the program $S_1' \cup S_2 = \{\gamma_1\}$. The predicate *negprop* has an empty definition in this program. Thus, by applying the negative unfolding rule R3 to clause γ_1 w.r.t. $\neg negprop$, we replace γ_1 by the following clause:

γ_8: $prop \leftarrow$

Thus, the final specialized program is $P_{sp} = \{prop \leftarrow\}$ and, as desired, we have proved that $\mathcal{K}, \langle a, 0\rangle \models \neg AF(neg)$.

6. Examples of Verification via Specialization

In this section we present some examples of verification of reactive systems by using our method based on program specialization. We have verified properties of the following reactive systems: (i) the Bakery protocol [35], (ii) the Ticket protocol [3], (iii) a Petri net with reset arcs [7], and (iv) several parameterized cache coherence protocols [30]. The verification of the various properties was performed automatically by using the experimental constraint logic program transformation system MAP [44].

Let us first make the following two preliminary remarks about our verification examples.

Remark 1
The reactive systems considered in this section are modeled by using linear equations and inequations over non-negative integers. However, our verification technique has been implemented in a constraint logic programming system which provides a solver for linear equations and inequations over rational numbers (see Section 6.5). We now show that the use of the solver for rational numbers provided by the system is correct for our verification examples.

Suppose that we want to verify that a temporal formula φ holds in a constraint-based Kripke structure $\mathcal{K}$, and let a state of $\mathcal{K}$ be represented by the $(k+m)$-tuple of variables $\langle S_1, \ldots, S_k, X_1, \ldots, X_m\rangle$, where we have singled out the variables $X_1, \ldots, X_m$ which range over non-negative integers. Now let us consider the constraint-based Kripke structure $\mathcal{Q}$ which is equal to $\mathcal{K}$ except that, for a state $\langle S_1, \ldots, S_k, X_1, \ldots, X_m\rangle$, the valuation of the variables $X_1, \ldots, X_m$ ranges over rational numbers. For the encoding of the temporal property φ, instead of program $P_{\mathcal{K}}[\varphi] = P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, we consider the program $P_{\mathcal{Q}}^{nat}[\varphi] = P_{\mathcal{Q}} \cup \{\gamma_1, \gamma_2^{nat}, \nu_1, \nu_2\}$, where γ_2^{nat} is the following clause:

γ_2^{nat}: $negprop \leftarrow nat(X_1) \wedge \ldots \wedge nat(X_m) \wedge sat(\langle S_1, \ldots, S_k, X_1, \ldots, X_m\rangle, init \wedge \neg\varphi)$

and ν_1 and ν_2 are the clauses that define the predicate *nat*:

ν_1: $nat(0) \leftarrow$
ν_2: $nat(Y) \leftarrow Y = X+1 \wedge nat(X)$

It can be shown that in all verification examples we will consider later in this section, the following property holds:

Property N

$$M(P_{\mathcal{Q}}^{nat}[\varphi]) \models \forall(\ nat(X_1) \wedge \ldots \wedge nat(X_m) \wedge t(\langle S_1, \ldots, S_k, X_1, \ldots, X_m\rangle, \langle T_1, \ldots, T_k, Y_1, \ldots, Y_m\rangle) \rightarrow nat(Y_1) \wedge \ldots \wedge nat(Y_m))$$

where t is a predicate interpreted as the transition relation in the structure $\mathcal{Q}$. Property N tells us that if in a state $\langle S_1, \ldots, S_k, X_1, \ldots, X_m\rangle$ the components $X_1, \ldots, X_m$ are non-negative integers,

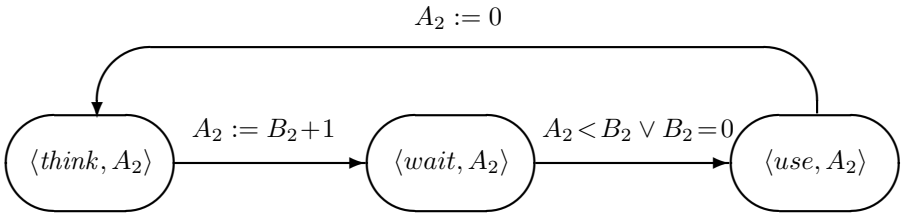


Figure 2: The Bakery Protocol: the transition relation R_α for process α. It depends also on the counter B_2 of process β.

then in the successor state $\langle T_1, \ldots, T_k, Y_1, \ldots, Y_m \rangle$ the components $Y_1, \ldots, Y_m$ are non-negative integers. Property N guarantees that $prop \in M(P_{\mathcal{K}}[\varphi])$ if and only if $prop \in M(P^{nat}_{\mathcal{Q}}[\varphi])$, where $prop$ is the predicate defined by the clause γ_1: $prop \leftarrow \neg negprop$. In our verification examples we have exploited this property and we have used program $P^{nat}_{\mathcal{Q}}[\varphi]$ together with a constraint solver for linear equations and inequations over the rational numbers.

Remark 2
In this section, when presenting formal specifications of reactive systems, we slightly depart from the definition of a constraint-based Kripke structure given in Section 3 (see Definition 5). In order to illustrate this point we need the following definition. We say that a state s is *reachable* if there exists a finite sequence $s_0, s_1, \ldots, s_n$ of states such that: (i) s_0 is an initial state, (ii) for $i = 0, \ldots, n-1$, state s_{i+1} is a successor state of s_i, and (iii) state s_n is s. In the examples of this section we assume that the carrier of the constraint interpretation coincides with the set of reachable states. By making this assumption we can write simpler specifications, because we may not define the transitions which start from unreachable states without loosing the totality property of the transition relation in accordance with Point 3 of Definition 5.

6.1. The Bakery Protocol

The Bakery Protocol ensures mutual exclusion between two concurrent processes that try to access a shared resource. Let α and β be the two concurrent processes we consider. The state of process α is represented by a pair $\langle A_1, A_2 \rangle$, where A_1 is an element of the set $\{think, wait, use\}$ of *control states*, and A_2 is a *counter* that takes values over the nonnegative integers. The state of process β is represented by a pair $\langle B_1, B_2 \rangle$ defined in a similar way. The time evolution of process α is modeled by the transition relation R_α depicted in Figure 2. This relation is defined by the following formula which is the disjunction of four events:

$$
\begin{array}{l}
t_\alpha(\langle A_1, A_2 \rangle, \langle A'_1, A'_2 \rangle, B_2) =_{def} \\
\qquad \begin{array}{llllll}
(A_1 = think & & \wedge\ A'_1 = wait & \wedge\ A'_2 = B_2{+}1) & \vee \\
(A_1 = wait & \wedge\ A_2 < B_2 & \wedge\ A'_1 = use & \wedge\ A'_2 = A_2) & \vee \\
(A_1 = wait & \wedge\ B_2 = 0 & \wedge\ A'_1 = use & \wedge\ A'_2 = A_2) & \vee \\
(A_1 = use & & \wedge\ A'_1 = think & \wedge\ A'_2 = 0) &
\end{array}
\end{array}
$$

The time evolution of process β is modeled by an analogous transition relation R_β. It is defined by the disjunction $t_\beta(\langle B_1, B_2 \rangle, \langle B'_1, B'_2 \rangle, A_2)$ obtained from $t_\alpha(\langle A_1, A_2 \rangle, \langle A'_1, A'_2 \rangle, B_2)$ by interchanging A_i with B_i and A'_i with B'_i, for $i = 1, 2$.

The state of the system resulting by the parallel composition of processes α and β, is represented by the 4-tuple $\langle A_1, A_2, B_1, B_2 \rangle$. The transition relation is defined by the following formula:

$$t(\langle A_1, A_2, B_1, B_2\rangle,\ \langle A_1', A_2', B_1', B_2'\rangle) =_{def}$$
$$(t_\alpha(\langle A_1, A_2\rangle, \langle A_1', A_2'\rangle, B_2) \wedge B_1' = B_1 \wedge B_2' = B_2) \quad \vee$$
$$(t_\beta(\langle B_1, B_2\rangle, \langle B_1', B_2'\rangle, A_2) \wedge A_1' = A_1 \wedge A_2' = A_2)$$

which, by distributing conjunction over disjunction, can be transformed into a disjunction of eight events.

This system has an infinite number of states, because there is no upper bound for the value of the counters, as illustrated by the underlined states of the following computation path:

$\langle think, 0, think, 0\rangle$, $\underline{\langle wait, 1, think, 0\rangle}$, $\langle wait, 1, wait, 2\rangle$, $\langle use, 1, wait, 2\rangle$,

$\langle think, 0, wait, 2\rangle$, $\langle think, 0, use, 2\rangle$, $\langle wait, 3, use, 2\rangle$, $\underline{\langle wait, 3, think, 0\rangle}$, ...

The set I of the initial states is a singleton specified by the constraint $init(\langle A_1, A_2,\ B_1, B_2\rangle)$ defined as follows:

$init(\langle A_1, A_2, B_1, B_2\rangle) =_{def} (A_1 = think \wedge A_2 = 0 \wedge B_1 = think \wedge B_2 = 0)$.

We have verified that the protocol indeed guarantees the mutually exclusive access to the resource. This property, also called *safety*, is expressed by the CTL formula $\neg EF(unsafe)$, where *unsafe* is an elementary property that holds if both processes are in the control state *use*, that is, the constraint $unsafe(\langle A_1, A_2, B_1, B_2\rangle)$ is defined as follows:

$unsafe(\langle A_1, A_2, B_1, B_2\rangle) =_{def} (A_1 = use \wedge B_1 = use)$

where A_2 and B_2 are any nonnegative integers.

We have also verified a *liveness* property ensuring that a process which requests the resource will eventually get it. For process α, liveness is expressed by the CTL formula $\neg EF(wait_\alpha \wedge \neg AF(use_\alpha))$, where the elementary properties $wait_\alpha$ and use_α are defined as follows:

$wait_\alpha(\langle A_1, A_2, B_1, B_2\rangle) =_{def} (A_1 = wait)$,

$use_\alpha(\langle A_1, A_2, B_1, B_2\rangle) =_{def} (A_1 = use)$

where A_2 and B_2 are nonnegative integers and B_1 is any element of $\{think, wait, use\}$.

6.2. The Ticket Protocol

Similarly to the Bakery Protocol, the Ticket Protocol provides a solution to the mutual exclusion problem. The interaction between the two processes α and β is controlled by a process γ that assigns tickets to α and β. The states of the processes α and β are represented as in the case of the Bakery Protocol. The state of process γ is represented by a pair $\langle T, N\rangle$ of nonnegative integers, where T is used for assigning a new ticket to α or β, and N provides an upper bound for the value of the ticket required for using the shared resource.

The transition relation $R_{\alpha|\gamma}$ for the parallel composition of $\alpha \mid \gamma$ of the processes α and γ (depicted in Figure 3), is defined by the following disjunction of three events:

$t_{\alpha|\gamma}(\langle A_1, A_2, T, N\rangle, \langle A_1', A_2', T', N'\rangle) =_{def}$

$(A_1 = think \wedge \quad A_1' = wait \wedge A_2' = T \wedge T' = T+1 \wedge N' = N) \quad \vee$

$(A_1 = wait \wedge A_2 \leq N \wedge A_1' = use \wedge A_2' = A_2 \wedge T' = T \wedge N' = N) \quad \vee$

$(A_1 = use \wedge \quad A_1' = think \wedge A_2' = 0 \wedge T' = T \wedge N' = N+1)$

The definition of the transition relation $R_{\beta|\gamma}$ for the parallel composition $\beta \mid \gamma$ of the processes β and γ is obtained from $t_{\alpha|\gamma}$ by replacing A_1 by B_1 and A_2 by B_2.

The state of the overall system is represented by the 6-tuple $\langle A_1, A_2, B_1,\ B_2, T, N\rangle$ and its transition relation R is defined by the following formula:

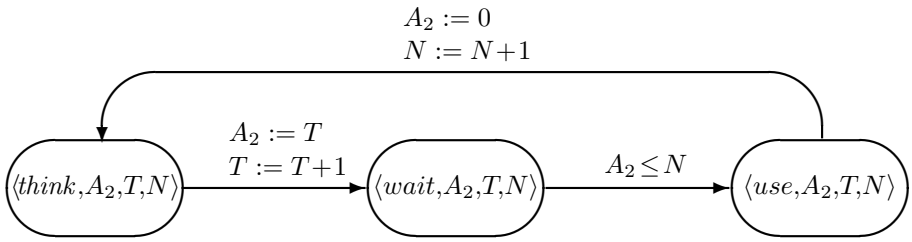


Figure 3: The Ticket Protocol: the transition relation $R_{\alpha|\gamma}$ for the parallel composition $\alpha \,|\, \gamma$ of the processes α and γ.

$$t(\langle A_1, A_2, B_1, B_2, T, N\rangle, \langle A_1', A_2', B_1', B_2', T', N'\rangle) =_{def}$$
$$(t_{\alpha|\gamma}(\langle A_1, A_2, T, N\rangle, \langle A_1', A_2', T', N'\rangle) \wedge B_1 = B_1' \wedge B_2 = B_2') \quad \vee$$
$$(t_{\beta|\gamma}(\langle B_1, B_2, T, N\rangle, \langle B_1', B_2', T', N'\rangle) \wedge A_1 = A_1' \wedge A_2 = A_2')$$

which, by distributing conjunction over disjunction, can be transformed into a disjunction of six events.

This system has an infinite number of states, because there is no upper bound for the values of T and N. The set I of the initial states is defined by the constraint $init(\langle A_1, A_2, B_1, B_2, T, N\rangle)$ as follows:

$init(\langle A_1, A_2, B_1, B_2, T, N\rangle) =_{def} (A_1 = think \wedge A_2 = 0 \wedge B_1 = think \wedge B_2 = 0 \wedge T = N)$

We have applied our verification method for proving the safety and the liveness properties of the Ticket Protocol. The safety property is expressed by the CTL formula $\neg EF\, unsafe$, where the elementary property *unsafe* is defined by the constraint $unsafe(\langle A_1, A_2, B_1, B_2, T, N\rangle)$ as follows:

$unsafe(\langle A_1, A_2, B_1, B_2, T, N\rangle) =_{def} (A_1 = use \wedge B_1 = use)$

where A_2, B_2, T, and N are nonnegative integers.

The starvation freedom property for a process, say process α, is expressed by the CTL formula $\neg EF(wait_\alpha \wedge \neg AF\, use_\alpha)$, where the elementary properties $wait_\alpha$ and use_α are defined as follows:

$wait_\alpha(\langle A_1, A_2, B_1, B_2, T, N\rangle) =_{def} (A_1 = wait)$, and

$use_\alpha(\langle A_1, A_2, B_1, B_2, T, N\rangle) =_{def} (A_1 = use)$

where A_2, B_2, T, and N are nonnegative integers and B_1 is any element of $\{think, wait, use\}$.

6.3. Reset Petri Nets

Reset Petri nets are Petri nets [56] augmented with *reset arcs* from places to transitions. We will not provide the general definition of this kind of Petri nets, but we will illustrate them through an example which is a variant of an example in [38].

Let us consider the reset Petri net shown in Figure 4, with the two *places* S_1 and S_2 depicted as circles, the two *transitions* t_1 and t_2 depicted as rectangles, the *flow relation* depicted as labeled arrows, and a reset arc from S_1 to t_1 depicted as an arrow with double head. A state, or *marking*, of the net is given by associating with each place a nonnegative integer which denotes the number of *tokens* residing in it. The initial state consists of one token in S_1 (depicted as a dot in the circle S_1 in Figure 4) and zero tokens in S_2. The state of the net changes according to the following two rules:

(T1) If in S_1 there is at least one token, then transition t_1 *fires*, and (i) the number of tokens in S_1 is reset to zero and (ii) two tokens are added to S_2;

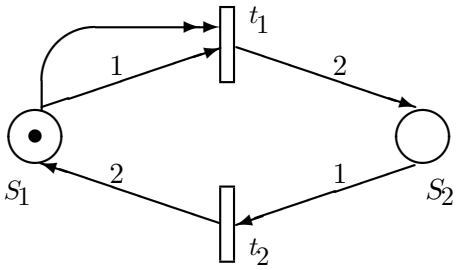


Figure 4: A Petri net with reset arcs.

(T2) If in S_2 there is at least one token, then transition t_2 fires, and (i) one token is taken away from S_2 and (ii) two tokens are added to S_1.

Note that, unlike the net presented as a working example in [38], the net of Figure 4 is not *bounded*, that is, there is no maximal number of tokens that can reside in a state. In other words, our net has infinitely many states.

Let us now apply our verification method for proving that the reset Petri net of Figure 4 never reaches a state where three tokens are in S_1 and no token is in S_2. The state of our net is represented by a pair $\langle N_1, N_2 \rangle$, where, for $i \in \{1, 2\}$, N_i is the number of tokens which are in place S_i. The transition relation R, which encodes the two rules T1 and T2, is defined by the disjunction of the following two events:

$t_1(\langle N_1, N_2 \rangle, \langle N_1', N_2' \rangle) =_{def} N_1 \geq 1 \wedge N_2 \geq 0 \wedge N_1' = 0 \wedge N_2' = N_2 + 2$
$t_2(\langle N_1, N_2 \rangle, \langle N_1', N_2' \rangle) =_{def} N_1 \geq 0 \wedge N_2 \geq 1 \wedge N_1' = N_1' + 2 \wedge N_2' = N_2 - 1$

The set I of the initial states is defined by the constraint $init(\langle N_1, N_2 \rangle)$ as follows:

$init(\langle N_1, N_2 \rangle) =_{def} (N_1 = 1 \wedge N_2 = 0)$

We have applied our verification method to prove that the state $\langle 3, 0 \rangle$ cannot be reached from the initial state. This *safety* property is expressed by the CTL formula $\neg EF\, p_{30}$ where the elementary property p_{30} holds in the state $\langle 3, 0 \rangle$ only, that is:

$p_{30}(\langle N_1, N_2 \rangle) =_{def} (N_1 = 3 \wedge N_2 = 0)$.

6.4. Parameterized Cache Coherence Protocols

Shared-memory multiprocessing systems make use of *local caches* associated with processors, for improving the efficiency of main memory access [30]. Every processor can modify the content of its local cache and, through a bus which is accessed in a mutually exclusive fashion, it can modify the content of the corresponding locations in main memory, and also modify the state of all local caches. As a consequence of *write* operations, the various local caches may hold inconsistent data. A *cache coherence protocol* guarantees data consistency, that is, it ensures that a *read* operation issued by a processor, for any given location, gets from its local cache the most recent data written on that location by any of the processors. Thus, a cache coherence protocol guarantees that, from a logical point of view, the system works as if each processor performs, in a mutually exclusive way, *read* and *write* operations directly on the shared main memory. We have verified properties of the following cache coherence protocols: Berkeley RISC, DEC Firefly, IEEE Futurebus+, Illinois University, MESI, MOESI, Synapse N+1, and Xerox

PARC Dragon. We have considered parameterized versions of these protocols, that is, protocols designed for an arbitrary number of processors with their associated local caches.

Here we present only the verification of the Synapse protocol for the N+1 computer. The verification of the other protocols is similar and for them we present the results of our experiments in Section 6.5. For a formal specification of all cache coherence protocols considered in this paper we refer to [17, 18].

For the description of the Synapse protocol, we make the following simplifying assumptions: (1) the system has a single bus and each processor has a single local cache, which is connected to the bus, (2) all local caches correspond to the same unique memory location, (3) the protocol is independent of the values stored in the local caches and in main memory. A cache can be in one of the following three states: *valid*, *dirty*, and *invalid*. If a cache is in state *valid*, then it holds the most recent data written by any processor and this data is also stored in main memory (more than one cache is allowed to be in state *valid*). If a cache is in state *dirty*, then it holds, in an exclusive way among all caches, the most recent data written by any processor (at most one cache is allowed to be in state *dirty*, and if a cache is in state *dirty* then all other caches are in state *invalid*). If a cache is in state *invalid*, then we cannot say whether it holds the most recent data written by any processor, and we cannot say whether it holds the data which is also stored in main memory.

We now describe the five rules of the protocol for a system of n processors and n caches, $c_1, \ldots, c_n$. Each rule should be regarded as atomic.

When a processor issues a *read* command to its cache c_i, one of the following two rules is applied.

Rule 1: READ HIT. If c_i is in state *valid* or *dirty*, then c_i returns its content and does not change its state.

Rule 2: READ MISS. If c_i is in state *invalid*, then the following actions take place sequentially:

(2.1) c_i issues a *public read* command to the bus;

(2.2) for all $j \neq i$, when a cache c_j receives the *public read* command from the bus, then: (2.2.1) if c_j is in state *dirty*, then it writes its content to main memory and goes to state *invalid*, else (2.2.2) if c_j is in state *valid* or *invalid*, no action is performed and, in particular, c_j does not change its state;

(2.3) c_i updates its content from main memory, and goes to state *valid*;

When a processor issues a *write* command to its cache, one of the following three rules is applied.

Rule 3: WRITE HIT 1. If the cache c_i is in state *dirty*, then c_i stores the new data and does not change its state.

Rule 4: WRITE HIT 2. If the cache c_i is in state *valid*, then the following actions take place sequentially:

(4.1) c_i issues a *private read* command to the bus;

(4.2) for all $j \neq i$, when a cache c_j receives the *private read* command from the bus, then: (4.2.1) if c_j is in state *dirty*, then it writes its content to main memory and goes to state *invalid*, else (4.2.2) if c_j is in state *valid* or *invalid*, no memory change is performed and c_j goes to state *invalid*.

(4.3) c_i stores the new data and changes its state to *dirty*;

Rule 5: WRITE MISS. If the cache c_i is in state *invalid*, then the following actions take place sequentially:

(5.1) c_i issues a *private read* command to the bus;

(5.2) all other caches react to the *private read* command received from the bus as described at Point (4.2);

(5.3) c_i updates its content from main memory and goes to state *dirty*.

It will be shown that Case 4.2.1 never occurs when applying Rule 4, because the protocol ensures that it is never the case that there is a cache in state *valid* and a cache in state *dirty* (see the property $\neg EF(unsafe2)$ below). However, we have not exploited this property in the above description of the protocol.

The state of the parameterized Synapse protocol is the collection of the states of each cache. In the initial state each cache is *invalid*. The data consistency property for the protocol is the following: it is impossible to reach a state where either (1) there are two or more *dirty* caches or (2) there are one or more *dirty* caches and one or more *valid* caches (see the elementary properties *unsafe*1 and *unsafe*2 below, respectively).

In order to verify data consistency, we consider an abstraction of the protocol, called *counting abstraction* [18], which consists in representing each state as a triple $\langle V, D, I\rangle$ of non-negative integers, where V, D, and I are the numbers of caches in state *valid*, *dirty*, and *invalid*, respectively. The transition relation which encodes Rules 1–5 for the abstracted protocol is the following:

$$
\begin{array}{lll}
t(\langle V,D,I\rangle,\langle V',D',I'\rangle) =_{def} & (V+D\geq 1 \wedge V'=V & \wedge D'=D \wedge I'=I) \vee \\
& (I\geq 1 \wedge V'=V+1 & \wedge D'=0 \wedge I'=D+I-1) \vee \\
& (D\geq 1 \wedge V'=V & \wedge D'=D \wedge I'=I) \vee \\
& (V\geq 1 \wedge V'=0 & \wedge D'=1 \wedge I'=D+V+I-1) \vee \\
& (I\geq 1 \wedge V'=0 & \wedge D'=1 \wedge I'=D+V+I-1)
\end{array}
$$

where each disjunct corresponds to a rule.

The set of the initial states is given by the constraint $init(\langle V, D, I\rangle)$ defined as follows:

$init(\langle V, D, I\rangle) =_{def} (V = 0 \wedge D = 0 \wedge I \geq 1)$

We have applied our method to prove the data consistency properties expressed by the CTL formulas $\neg EF(unsafe1)$ and $\neg EF(unsafe2)$, where the elementary properties *unsafe*1 and *unsafe*2 are defined by constraints as follows:

$unsafe1(\langle V, D, I\rangle) =_{def} (D \geq 2)$, and

$unsafe2(\langle V, D, I\rangle) =_{def} (D \geq 1 \wedge V \geq 1)$.

6.5. Experimental Results

All verification examples presented in Sections 6.1 – 6.4 have been developed in a fully automatic way by using our experimental transformation system MAP [44] for constraint logic programs. We have also verified a version of the Bakery protocol for three processes and the various cache coherence protocols mentioned in Section 6.4. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the SICStus Prolog `clpq` library to solve constraints (see Table 1).

We have performed the same verification experiments by using the DMC system [19] and the HyTech system, version 1.04f [31] (see Table 1). All experiments have been conducted on an Intel Pentium M740, 1.73 GHz under the Linux operating system.

Protocol	Property	MAP	DMC (no Abs.)	DMC (Abs.)	HyTech
Bakery (2 processes)	*safety*	0.05	0.01	0.05	0.02
(mutual exclusion)	*liveness*	0.13	0.10	0.10	–
Bakery (3 processes)	*safety*	0.37	0.71	4.95	0.55
(mutual exclusion)					
Ticket	*safety*	0.20	↑	0.09	↑
(mutual exclusion)	*liveness*	0.39	↑	0.34	–
Reset Petri Net	*safety*	0.08	≤0.005	≤0.005	≤0.005
Synapse N+1	*safety*	0.04	≤0.005	≤0.005	0.01
(cache coherence)					
Berkeley RISC	*safety*	0.07	0.04	0.05	0.01
(cache coherence)					
Xerox Dragon	*safety*	0.07	0.12	0.12	0.04
(cache coherence)					
DEC Firefly	*safety*	0.05	0.08	0.13	0.03
(cache coherence)					
IEEE Futurebus+	*safety*	0.22	7.41	15.97	0.63
(cache coherence)					
Illinois University	*safety*	0.06	0.11	0.18	0.03
(cache coherence)					
MESI	*safety*	0.07	0.06	0.10	0.02
(cache coherence)					
MOESI	*safety*	0.11	0.08	0.14	0.02
(cache coherence)					

Table 1: Experimental verification results. Times are expressed in seconds.

In Table 1, the *safety* and *liveness* properties of the Bakery protocol for two processes are defined as indicated in Section 6.1. The *safety* property of the Bakery protocol for three processes is defined similarly to the same property for two processes. The *safety* and *liveness* properties of the Ticket protocol are defined as indicated in Section 6.2. The *safety* property of the reset Petri net is defined, as indicated in Section 6.3, by the CTL formula $\neg EF\,(p_{30})$, where p_{30} is the elementary property that holds in the state $\langle N_1, N_2\rangle$ iff $N_1 = 3 \wedge N_2 = 0$.

The *safety* property of the Synapse protocol is defined by the CTL formula $\neg EF\,(\mathit{unsafe}1 \vee \mathit{unsafe}2)$, where $\mathit{unsafe}1$ and $\mathit{unsafe}2$ are the elementary properties defined at the end of Section 6.4. For the other cache coherence protocols the specification of the various *safety* properties can be found in [17, 18].

From the third to the sixth column of Table 1 we have reported the times needed to verify the various properties by using MAP, DMC, and HyTech. For the DMC system we have two columns: in DMC (no Abs.) and DMC (Abs.) we have reported the running times obtained by using DMC without and with an abstraction operator, respectively.

In Table 1 the symbol "↑" means "nontermination within 10 minutes", and the symbol "−" indicates that the test has not been performed. Note that the liveness properties have not been tested by using HyTech (see the entries "−" in the last column), because it has no specific primitives for encoding such properties, while the *safety* properties have been encoded by using the *reach backward* primitive provided by that system. HyTech diverges on all our examples when the *safety* properties are encoded by using the *reach forward* primitive.

No abstraction techniques have been used to approximate the computation of the fixpoints in the examples reported in column DMC (no Abs.) and HyTech. This explains the nontermination observed for the Ticket example. By comparing the columns DMC (no Abs.) and DMC (Abs.) it can be noted that the use of abstraction improves termination at the expense of an increase of the verification time.

In the experiments we have performed, MAP is less efficient than DMC and HyTech when verifying *safety* properties in simple protocols, while it performs better on the two most complex protocols, that is, the Bakery protocol for three processes and the IEEE Futurebus+ protocol. In conclusion, we may say that the performance of our system is comparable with the one of DMC and HyTech.

The encodings of the various reactive systems reported in Table 1 are available from `http://www.iasi.rm.cnr.it/~proietti/system.html`.

7. Witnesses and Counterexamples

One of the most important features of the model checking techniques for finite state systems is the ability to find *witnesses* and *counterexamples* [12]. A witness of a formula with an existential path quantifier is a computation path which shows that the formula holds, and a counterexample of a formula with a universal path quantifier is a computation path which shows that the formula does not hold. In this section, we demonstrate, by means of examples, how to enhance our verification technique for infinite state systems so that witnesses and counterexamples are generated. In particular, we consider the reactive system presented in Example 3 of Section 3 (and continued in Section 5.3), and we show how to generate a witness of a formula of the form $EU(\varphi_1, \varphi_2)$ and how to generate a counterexample of a formula of the form $AF\,\varphi$. The case of formulas with the operator EX is simpler and we leave it to the reader.

The basic idea consists in adding an extra argument to the *sat* predicate defined by the Encoding Program $P_{\mathcal{K}}$. This extra argument keeps track of the transitions connecting the

states to their successor states. Note that, by our assumptions on the transition relation (see Definition 5), a sequence of transitions and an initial state uniquely determine a sequence of states.

Example 8. [Generating Witnesses] Let us consider the reactive system depicted in Figure 1 (see Example 3 of Section 3). Suppose that we want to generate a witness of a formula of the form $EU\,(\varphi_1, \varphi_2)$ starting from the initial state, that is, a sequence of transitions to be applied from the initial state $\langle a, 0\rangle$ so to get to a state where φ_2 holds, passing through states where φ_1 holds. In order to do so we perform the following two steps. (Step 1) We modify the clauses of $P_{\mathcal{K}}$ which define the satisfiability of the formula $EU\,(\varphi_1, \varphi_2)$ by adding to the predicate *sat* an extra argument which is a witness of that formula. (Step 2) We generate a witness of $EU\,(\varphi_1, \varphi_2)$ by applying the specialization strategy of Section 5 to the program with the modified clauses.
Step 1. In order to modify the definition of satisfiability, we first modify the encoding of the transition relation by adding to $t(X, Y)$ an extra argument T which is the transition connecting state X to state Y. The resulting predicate $t_w(X, T, Y)$ is defined by the following three clauses, one for each transition t_1, t_2, and t_3 (recall that in this example every state is represented by a pair):

$t_w(\langle X_1, X_2\rangle, t_1, \langle Y_1, Y_2\rangle) \leftarrow X_1 = a \wedge Y_1 = a \wedge Y_2 = X_2 + 2$
$t_w(\langle X_1, X_2\rangle, t_2, \langle Y_1, Y_2\rangle) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_1 = b \wedge Y_2 = X_2$
$t_w(\langle X_1, X_2\rangle, t_3, \langle Y_1, Y_2\rangle) \leftarrow X_1 = b \wedge Y_1 = b \wedge Y_2 = X_2 + 1$

Next, we modify the clauses of the predicate *sat* relative to the *EU* operator by adding a third argument which is a witness of $EU\,(\varphi_1, \varphi_2)$ starting from a state X. By doing so we obtain a new predicate sat_w defined as follows:

$sat_w(X, eu(F_1, F_2), [\,]) \leftarrow sat(X, F_2)$
$sat_w(X, eu(F_1, F_2), [T|\,Ts]) \leftarrow sat(X, F_1) \wedge t_w(X, T, Y) \wedge sat_w(Y, eu(F_1, F_2), Ts)$

Suppose that we want to generate a witness of $EU(is\_a, is\_b \wedge geq4)$, where *is_a*, *is_b*, and *geq*4 are elementary properties such that the following constrained facts hold:

$sat(\langle X_1, X_2\rangle, is\_a) \leftarrow X_1 = a$
$sat(\langle X_1, X_2\rangle, is\_b) \leftarrow X_1 = b$
$sat(\langle X_1, X_2\rangle, geq4) \leftarrow X_2 \geq 4$

We introduce the clause:

γ_{weu}: $witness_{eu}(W) \leftarrow X_1 = a \wedge X_2 = 0 \wedge sat_w(\langle X_1, X_2\rangle, eu(is\_a, is\_b \wedge geq4), W)$

Clearly, $witness_{eu}(W)$ holds iff W is a witness of $EU(is\_a, is\_b \wedge geq4)$ starting from the initial state $\langle a, 0\rangle$.
Step 2. Now, we apply the *UDF* procedure starting from γ_{weu} and we get the following specialized program:

$witness_{eu}([t_1|W]) \leftarrow X_1 = a \wedge X_2 = 2 \wedge new1(\langle X_1, X_2\rangle, W)$
$new1(\langle X_1, X_2\rangle, [t_1|W]) \leftarrow X_1 = a \wedge X_2 \geq 0 \wedge Y_2 = X_2 + 2 \wedge new1(\langle X_1, Y_2\rangle, W)$
$new1(\langle X_1, X_2\rangle, [t_2|W]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_1 = b \wedge new2(\langle Y_1, X_2\rangle, W)$
$new2(\langle X_1, X_2\rangle, [\,]) \leftarrow X_1 = b \wedge X_2 \geq 4$

By repeatedly applying the unfolding rule we get, among other clauses, the following one:

$witness_{eu}([t_1, t_1, t_2]) \leftarrow$

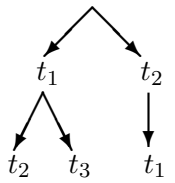
which shows that a witness of the formula $EU(is\_a, is\_b \wedge geq4)$ is the sequence $[t_1, t_1, t_2]$ of transitions. □

Now we show an example of generation of a counterexample for a formula $AF\,\varphi$ that does not hold.

Example 9. [Generating Counterexamples] Let us consider again the reactive system presented in Example 3 of Section 3. Suppose that we want to generate a counterexample of the formula $AF\ neg$, which does not hold for the reactive system under consideration (see Section 5.3). Similarly to the previous Example 8, we perform two steps. (Step 1) We modify the predicate $sat(X, af(F))$ by introducing a new predicate $sat_w(X, af(F), W)$, where the extra argument W encodes a witness of the formula $AF\,\varphi$ starting from X. That witness is a *tree of transitions* (rather than a *sequence of transitions*, as in the case of the witness of $EU(\varphi_1, \varphi_2)$) such that for every root-to-leaf path, if we apply the sequence of transitions corresponding to that path, starting from the state X we get to a state where φ holds. (Step 2) We generate the counterexamples of $AF\,\varphi$ by applying our specialization strategy to the program defining sat_w.
Step 1. The predicate sat_w, and the predicates sat_all_w and ts_w on which sat_w depends, are defined as follows:

$sat_w(X, af(F), [\,]) \leftarrow sat(X, F)$
$sat_w(X, af(F), W) \leftarrow ts_w(X, TYs) \wedge sat\_all_w(TYs, af(F), W)$
$sat\_all_w([\,], F, [\,]) \leftarrow$
$sat\_all_w([(T, Y)\,|\,TYs], F, [(T, W)\,|\,Ws]) \leftarrow sat_w(Y, F, W) \wedge sat\_all_w(TYs, F, Ws)$
$ts_w(\langle X_1, X_2\rangle, [(t_1, \langle Y_1, Y_2\rangle)]) \leftarrow X_1 = a \wedge X_2 \leq 0 \wedge Y_1 = a \wedge Y_2 = X_2 + 2$
$ts_w(\langle X_1, X_2\rangle, [(t_1, \langle Y_{11}, Y_{12}\rangle), (t_2, \langle Y_{21}, Y_{22}\rangle)]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_{11} = a \wedge Y_{12} = X_2 + 2 \wedge$
$\qquad\qquad Y_{21} = b \wedge Y_{22} = X_2$
$ts_w(\langle X_1, X_2\rangle, [(t_3, \langle Y_1, Y_2\rangle)]) \leftarrow X_1 = b \wedge Y_1 = b \wedge Y_2 = X_2 + 1$

In the above program $ts_w(X, [(tr_1, Z_1), \ldots, (tr_n, Z_n)])$ holds iff $tr_1, \ldots, tr_n$ are *all* the transitions starting from state X, where, for $i = 1, \ldots, n$, tr_i belongs to $\{t_1, t_2, t_3\}$ and from state X by applying tr_i we get to the successor state Z_i. Note that the last argument W of sat_w is a term encoding a tree of transitions which is a witness of $AF\,\varphi$. For instance, the tree of transitions



is encoded by the term $[(t_1, [(t_2, [\,]), (t_3, [\,])],\ (t_2, [(t_1, [\,])])]$.

Now we introduce the new clause:

γ_{waf}: $witness_{af}(W) \leftarrow X_1 = a \wedge X_2 = 0 \wedge sat_w(\langle X_1, X_2\rangle, initial \wedge af(neg), W)$

We have that $witness_{af}(W)$ holds iff W is a witness of $AF\ neg$ starting from the initial state $\langle a, 0\rangle$.
Step 2. By applying the *UDF* procedure starting from γ_{waf} we get the following specialized program P_W:

1. $witness_{af}([(t_1, W)]) \leftarrow X_1 = a \wedge X_2 = 2 \wedge new1(\langle X_1, X_2\rangle, W)$
2. $new1(\langle X_1, X_2\rangle, [(t_1, W)]) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_2 = 2 \wedge new1(\langle X_1, Y_2\rangle, W)$
3. $new1(\langle X_1, X_2\rangle, [(t_1, W_1), (t_2, W_2)]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_2 = X_2 + 2 \wedge Y_3 = b \wedge$
$\qquad new1(\langle X_1, Y_2\rangle, W_1) \wedge new2(\langle Y_3, X_2\rangle, W_2)$
4. $new2(\langle X_1, X_2\rangle, [(t_3, W)]) \leftarrow X_1 = b \wedge X_2 > 0 \wedge Y_2 = X_2 + 1 \wedge new2(\langle X_1, Y_2\rangle, W)$

All predicates in P_W are useless (see Definition 4) and all clauses of P_W are removed by the RU procedure (see Section 5.2.2), thereby proving that $witness_{af}(W)$ does *not* hold for any value of W. Indeed, the execution of P_W does not terminate for the goal $witness_{af}(W)$ and generates an infinite tree for W. Now, we have that the maximal paths of that infinite tree are exactly all the counterexamples of $AF\, neg$ (that is, the witnesses of $EG\, \neg neg$). These paths can be generated by the following program P_C:

1.1 $counterexample_{af}([t_1|W]) \leftarrow X_1 = a \wedge X_2 = 2 \wedge c1(\langle X_1, X_2\rangle, W)$
2.1 $c1(\langle X_1, X_2\rangle, [t_1|W]) \leftarrow X_1 = a \wedge X_2 = 0 \wedge Y_2 = 2 \wedge c1(\langle X_1, Y_2\rangle, W)$
3.1 $c1(\langle X_1, X_2\rangle, [t_1|W_1]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_2 = X_2 + 2 \wedge c1(\langle X_1, Y_2\rangle, W_1)$
3.2 $c1(\langle X_1, X_2\rangle, [t_2|W_2]) \leftarrow X_1 = a \wedge X_2 > 0 \wedge Y_3 = b \wedge c2(\langle Y_3, X_2\rangle, W_2)$
4.1 $c2(\langle X_1, X_2\rangle, [t_3|W]) \leftarrow X_1 = b \wedge X_2 > 0 \wedge Y_2 = X_2 + 1 \wedge c2(\langle X_1, Y_2\rangle, W)$

Each clause of P_C has been obtained from a clause γ of P_W by: (i) renaming predicates, (ii) selecting a branch of the tree in the head of γ, (iii) selecting the corresponding atom in the body of γ, and (iv) applying the *solve* function to the constraint of γ w.r.t. the set of variables which occur either in the head of γ or in the atom of γ selected at Point (iii).

For instance, clause 3.1 has been obtained from clause 3 by: (i) renaming $new1$ to $c1$, (ii) selecting the branch $[t_1|W_1]$ from the tree $[(t_1, W_1), (t_2, W_2)]$ occurring in the head of clause 3, (iii) selecting the atom $new1(W_1, \langle X_1, Y_2\rangle)$, with $new1$ renamed to $c1$, and (iv) computing $solve(X_1 = a \wedge X_2 > 0 \wedge Y_2 = X_2 + 2 \wedge Y_3 = b, \{X_1, X_2, Y_2\})$. We can generate all *finite prefixes* of counterexamples of $AF\, neg$ by adding to P_C the facts: $c1(X, [\,]) \leftarrow$ and $c2(X, [\,]) \leftarrow$.

Finally, looking at program P_C, we have that the counterexamples of $AF\, neg$ are the infinite sequences belonging to the regular ω-language $\{t_1^\omega, t_1 t_2 t_3^\omega\}$ [67]. Note, however, that in the case of an arbitrary infinite state system, the set of counterexamples of $AF\, \varphi$ is *not* a regular ω-language, because otherwise $AF\, \varphi$ would be a decidable property. □

8. Related Work and Conclusions

We have presented a method based on constraint logic programming and program transformation to perform model checking of infinite state concurrent systems. The main features of our method are: (i) the representation of an infinite state concurrent system by means of constraints over the set of states, and the encoding of the temporal properties of the system, expressed by CTL formulas, as general logic programs with the perfect model semantics, (ii) the application of a rule-based specialization strategy for verifying CTL formulas, and (iii) the use of a generalization strategy to ensure that our verification method terminates in all cases.

The main motivation for developing our approach is that it allows us to transfer techniques and tools for constraint solving, logic programming, and program transformation, to the field of infinite state model checking. Moreover, due to the generality of the methodologies we propose, our verification method could easily be adapted to solve a wide variety of model checking problems.

Our approach shares with the techniques of *deductive model checking* (see, for instance, [48, 52, 63, 64]) the idea that first order formulas can be suitably used to describe infinite sets of states and logical inference rules can be applied to verify properties of systems. However, our method is characterized by the use of constraint logic programming and program transformation, which from a technical point of view make our approach very different from the approaches described in the above mentioned deductive model checking works.

Our method is specifically related to the methods that make use of logic programming and constraints as a basis for the verification of finite or infinite state concurrent systems [19, 24, 27, 28, 40, 38, 49, 55, 57, 58]. Let us now briefly compare these methods with our work.

The methods presented in [55] and [49] make use of logic programming and constraint logic programming, respectively, for the verification of *finite* state systems. In particular, in [55] the authors present XMC, a model checking system implemented in the tabulation-based logic programming language XSB [59]. XMC can verify temporal properties expressed in the alternation-free fragment of the μ-calculus of finite state concurrent systems specified in a CCS-like language. The XMC implementation contains many source-level optimizations that take advantage of the tabulation-based execution mechanism of XSB, thereby achieving performances comparable to those of the most advanced model checkers. The model checker presented in [49] can be applied to verify CTL properties of finite state systems by using CLP programs with constraints which are defined over finite domains and are closed under conjunction, disjunction, variable projection, and negation. The verification process is performed by executing a CLP program encoding the semantics of CTL in an extended execution model that uses constructive negation and tabled resolution.

Our method is more general than those presented in [55, 49], as we can verify properties of *infinite* state systems. Indeed, it could be shown that the case of finite state systems can be handled by our transformation strategy *without using any generalization technique*.

The use of constraint logic programs as a means of representing and reasoning about infinite state systems has been first advocated in [27, 28], where an automatic method for verifying safety properties of Petri nets with parametric initial markings is presented. By using this method the set of reachable markings of a Petri net is characterized as the least fixpoint of a suitable logic program with arithmetic constraints. The aim of the method is to express, if at all possible, the least fixpoint of this logic program as a Presburger formula, so that the reachability of a given marking can be checked. Invariant checking and transformations of Petri nets are used for improving performance.

In Section 6.3 we have shown through an example that our method can be used to prove safety properties of Petri nets. It should be noted that our method can also be applied to verify properties more complex than safety, such as liveness. We think that, however, in order to assess the viability of our approach for the verification of Petri nets, a more extensive experimentation is needed.

The method described in [19] is aimed at the verification of CTL properties of infinite state concurrent systems by using constraint logic programming. Depending on the formula and the system being verified, suitable definite CLP programs are introduced. CTL properties are then verified by computing exact and approximated least and greatest fixpoints of those programs.

The main differences between our method and the one presented in [19] are the following. (i) In order to specify concurrent systems and their temporal properties we use general logic programs with the perfect model semantics, instead of least and greatest fixpoints of definite logic programs, and thus our specifications are, in principle, executable by currently available logic programming systems. (ii) Our transformation strategy always returns a program which has the same perfect model as the initial one, and not an over or under approximation of that model; thus, our approach allows us to apply further analysis and transformation to the final program in the case where we are not able to complete the desired verification task in the first step. (iii) We use a generalization strategy which, unlike the technique presented in [19], always guarantees termination. We have shown in Section 6.5 that the performances achieved by the two methods are similar.

The approach presented in [40] combines partial deduction (that is, partial evaluation of logic programs) and abstract interpretation for the verification of properties of infinite state systems. This approach is applied to solve coverability problems of Petri nets with reset arcs and, in particular, to compute the Karp-Miller tree and Finkel's minimal coverability set [38].

The use of constraints makes our approach more powerful than the one proposed in [40]. Indeed, during program specialization we manipulate clauses by constraint solving, and this transformation rule is not available in partial deduction. Our specialization strategy also makes use of other transformation rules, such as negative unfolding (R3), removal of subsumed clauses (R6s), and removal of useless clauses (R6u), which are not considered by partial deduction. Finally, our generalization technique makes use of a constraint generalization operator which exploits the properties of the constraints (see Definition 15), while the *most specific generalization* used in [40] is based on syntactic term anti-unification.

Logic programming and program transformation have been applied in [57, 58] for proving safety and liveness properties of *parameterized* finite state systems with various network topologies. The verification process is carried out by proving equivalences of predicates defined by definite logic programs using unfold/fold transformations. Unlike our method, the verification technique of [57, 58] is not fully automatic, as it requires human intervention to invent suitable invariants during transformation.

The method presented in [24] allows the specification and verification of concurrent systems by means of a concurrent extension of constraint logic programming, called *timed concurrent constraint programming* (tccp). Similarly to our approach and to others mentioned above, this work makes use of constraints to specify infinite state systems. A model checking algorithm for tccp programs is obtained by restricting time to a finite interval given by the user, thereby reducing a potentially infinite state program to a finite state approximation. This approach seems to be less general and mechanical than ours, because in ours no time interval has to be fixed in advance.

The transformation rules for locally stratified constraint logic programs considered in this paper (see Section 4) are specialized versions of the unfold/fold rules presented in [26] and, indeed, their correctness with respect to the perfect model semantics is a consequence of some results proved in that paper.

The unfold/fold rules for definite CLP programs have been first introduced in [8, 23] by extending the rules for definite logic programs presented in [66]. If we consider definite CLP programs only, the definition and folding rules of [8, 23] are more general than those defined in Section 4, because they allow the body of a definition to be a non-atomic constrained goal. However, in Sections 5 and 6 we have shown that by allowing atomic definitions only, we are able to verify several interesting CTL properties of infinite state systems.

Unfold/fold rules for transforming stratified constraint logic programs have been first presented in [42]. However, the set of rules considered in [42] does not include the negative unfolding and negative folding rules (Rules R3 and R5, respectively, of Section 4). Moreover, the folding rule of [42] is not capable to derive recursive clauses, and the derivation of recursive clauses is a crucial feature of the method we propose in the present paper (see Section 5).

The rule for deleting useless clauses (Rule R6u of Section 4) is not present in [8, 23, 42].

The idea of using unfold/fold transformation strategies for theorem proving and, in particular, for proving properties of logic programs, has been explored in various papers (see, for instance, [50, 51, 57, 58]).

The unfold/fold-based specialization strategy presented in this paper is an extension to locally

stratified CLP programs of the strategy for specializing definite CLP programs presented in [25]. Note that, however, the specialization strategy of Section 5 is specifically designed to transform programs that encode properties of infinite state systems, while the specialization strategy of [25] is a general purpose strategy.

The specialization strategy presented in Section 5 is also related to other techniques for logic program specialization (see [29, 34, 37] for surveys) and, in particular, to specialization techniques based on unfold/fold rules [53, 60]. As already mentioned, the main feature of our strategy is that it is oriented to the specialization of locally stratified CLP programs that encode properties of infinite state systems.

In order to guarantee the termination of our specialization strategy, we make use of a generalization technique which extends to constraint logic programs other techniques, based on *well-quasi orderings*, employed in various methods for the specialization of logic programs [36, 39, 65]. To make this extension we have introduced a *generalization operator* which is inspired to the *widening operator* for linear constraints defined in the field of abstract interpretation [13]. However, as discussed in Section 5.2.1, our generalization operator is different from the widening operator. In fact, it could be shown that by applying our verification technique with the generalization operator replaced by the widening operator, we would fail to prove many of the properties considered in Section 6.

We have already mentioned that notions related to abstract interpretation have been used in other techniques for the verification of infinite state systems, such as those in [2, 16, 19, 68]. However, it should be noted that our generalization technique preserves the semantics of the program that encodes the infinite state system to be verified, while the techniques presented in [2, 16, 19, 68] compute approximated models of the system.

In conclusion, the present work contributes to demonstrate that the use of constraint logic programming as a modeling language together with program transformation as an inference device, provides a very flexible and powerful methodology for the verification of infinite state systems and, more generally, for proving properties of software systems. Indeed, we have shown that constraints allow simple representations of infinite sets of values, and the declarative nature of logic programming makes it easy to model a large variety of systems and properties. We have also shown that transformation-based proof methods are very general and powerful, and they can also be very efficient when tailored to the task of proving properties of infinite state systems.

Acknowledgements

We would like to thank Giorgio Delzanno, Sandro Etalle, and Michael Leuschel for very helpful comments on drafts of this work. Fulvio Forni and Valerio Senni helped us with the experiments presented in Section 6.5.

Appendix

Proof:[Proof of Theorem 3.3] By Lemma 2.2 it suffices to show that for all states $s \in D$ and CTL formulas φ, we have that: $\mathcal{K}, s \models \varphi$ iff $sat(s, \varphi) \in lfp(T_{P_{\mathcal{K}},\alpha})$, where $\alpha = size(\varphi)$. The proof is by structural induction on φ. By induction hypothesis we assume that, for all states $s \in D$ and for all proper subformulas ψ of φ,

$\mathcal{K}, s \models \psi$ iff $sat(s, \psi) \in lfp(T_{P_{\mathcal{K}},\beta})$ (IndHyp1)

where $\beta = size(\psi)$. Now we consider the following cases.

Case 1. (φ is the elementary property e) For all states $s \in D$ we have that:
$\mathcal{K}, s \models e$
iff $\mathcal{D} \models e(s)$ (by Point 4 of Definition 5)
iff $sat(s, e) \in \mathit{lfp}(T_{P_\mathcal{K},1})$ (by the definitions of $P_\mathcal{K}$ and $T_{P_\mathcal{K},1}$).

Case 2. (φ is $\neg\psi$) For all states $s \in D$ we have that:
$\mathcal{K}, s \models \neg\psi$
iff $\mathcal{K}, s \models \psi$ does not hold (by the definition of $\mathcal{K}, s \models \neg\psi$, see Section 2.3)
iff $sat(s, \psi) \notin \mathit{lfp}(T_{P_\mathcal{K},\beta})$, where $\beta = size(\psi)$ (by IndHyp1)
iff $sat(s, \neg\psi) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$, where $\alpha = \beta + 1 = size(\neg\psi)$
(by the definitions of $P_\mathcal{K}$ and $T_{P_\mathcal{K},\alpha}$).

Case 3. (φ is $\psi_1 \wedge \psi_2$) For all states $s \in D$ we have that:
$\mathcal{K}, s \models \psi_1 \wedge \psi_2$
iff $\mathcal{K}, s \models \psi_1$ and $\mathcal{K}, s \models \psi_2$
(by the definition of $\mathcal{K}, s \models \psi_1 \wedge \psi_2$, see Section 2.3)
iff $sat(s, \psi_1) \in \mathit{lfp}(T_{P_\mathcal{K},\beta 1})$ and $sat(s, \psi_2) \in \mathit{lfp}(T_{P_\mathcal{K},\beta 2})$
where $\beta 1 = size(\psi_1)$ and $\beta 2 = size(\psi_2)$ (by IndHyp1)
iff $sat(s, \psi_1 \wedge \psi_2) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$, where $\alpha = \beta 1 + \beta 2 + 1 = size(\psi_1 \wedge \psi_2)$
(by the definitions of $P_\mathcal{K}$ and $T_{P_\mathcal{K},\alpha}$).

Case 4. (φ is $EX(\psi)$) For all states $s \in D$ we have that:
$\mathcal{K}, s \models EX\,\psi$
iff there exists a state $s' \in D$ such that $s\,R\,s'$ and $\mathcal{K}, s' \models \psi$
(by the definition of $\mathcal{K}, s \models EX(\psi)$, see Section 2.3)
iff there exist a state $s' \in D$ and $j \in \{1, \ldots, k\}$ such that:
(i) $\mathcal{D} \models t_j(s, s')$ and (ii)$sat(s', \psi) \in \mathit{lfp}(T_{P_\mathcal{K},\beta})$, where $\beta = size(\psi)$
(by the definition of R and IndHyp1)
iff there exist a state $s' \in D$, a valuation v, and a clause $\gamma \in P_\mathcal{K}$ such that:
(i) $v(\gamma)$ is of the form $sat(s, ex(\psi)) \leftarrow t(s, s') \wedge sat(s', \psi)$,
(ii) $t(s, s') \in \mathit{lfp}(T_{P_\mathcal{K},0})$, and (iii) $sat(s', \psi) \in \mathit{lfp}(T_{P_\mathcal{K},\beta})$
(by the definitions of $P_\mathcal{K}$ and $T_{P_\mathcal{K},0}$)
iff $sat(s, ex(\psi)) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$, where $\alpha = \beta + 1 = size(ex(\psi))$
(by the definition of $T_{P_\mathcal{K},\alpha}$).

In the rest of the proof, given a formula ψ, the set $\{s \in D \,|\, \mathcal{K}, s \models \psi\}$, that is, the set of states of $\mathcal{K}$ in which ψ is true, is also denoted by $[\psi]$.

Case 5. (φ is $EU(\psi_1, \psi_2)$) Given a set of states Y, let $\tau_{EU}(Y)$ denote the set $[\psi_2] \cup ([\psi_1] \cap \{s \in D \,|\, \text{there exists } s' \in D \text{ such that } s\,R\,s' \text{ and } s' \in Y\})$. From [21] we have that $\mathcal{K}, s \models EU(\psi_1, \psi_2)$ holds iff $s \in \mathit{lfp}(\tau_{EU})$. Thus, we have to show that, for all states $s \in D$, $s \in \mathit{lfp}(\tau_{EU})$ iff $sat(s, eu(\psi_1, \psi_2)) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$, where $\alpha = size(eu(\psi_1, \psi_2))$.

We split the proof into two parts and we prove: (5.1) for all $h \geq 0$ and for all $s \in D$, $s \in \tau^h_{EU}(\emptyset)$ implies $sat(s, eu(\psi_1, \psi_2)) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$ and (5.2) for all $h \geq 0$ and for all $s \in D$, $sat(s, eu(\psi_1, \psi_2)) \in T^h_{P_\mathcal{K},\alpha}(\emptyset)$ implies $s \in \mathit{lfp}(\tau_{EU})$. We show the proof of (5.1) only. The proof of (5.2) is similar and is omitted.

We proceed by induction on h. The base case trivially holds because $\tau^0_{EU}(\emptyset) = \emptyset$. Now, we assume the following inductive hypothesis:

for all $s \in D$, $s \in \tau^h_{EU}(\emptyset)$ implies $sat(s, eu(\psi_1, \psi_2)) \in \mathit{lfp}(T_{P_\mathcal{K},\alpha})$ (IndHyp2)

where $\alpha = size(eu(\psi_1, \psi_2))$, and we prove that, for all $s \in D$, $s \in \tau^{h+1}_{EU}(\emptyset)$ implies $sat(s, eu(\psi_1, \psi_2)) \in lfp(T_{P_\mathcal{K},\alpha})$.
We have that:

$s \in \tau^{h+1}_{EU}(\emptyset)$

implies *either* $\mathcal{K}, s \models \psi_2$
or $\mathcal{K}, s \models \psi_1$ and there exists $s' \in D$ such that $s\,R\,s'$ and $s' \in \tau^h_{EU}(\emptyset)$
(by the definition of τ_{EU})

implies *either* $sat(s, \psi_2) \in lfp(T_{P_\mathcal{K},\beta 2})$
or $sat(s, \psi_1) \in lfp(T_{P_\mathcal{K},\beta 1})$ and there exists $s' \in D$ such that
$\mathcal{D} \models t_1(s, s') \vee \ldots \vee t_k(s, s')$ and $sat(s', eu(\psi_1, \psi_2)) \in lfp(T_{P_\mathcal{K},\alpha})$,
where $\beta 1 = size(\psi_1)$ and $\beta 2 = size(\psi_2)$
(by the definition of R and the inductive hypotheses IndHyp1 and IndHyp2)

implies that there exist a valuation v and a clause $\gamma \in P_\mathcal{K}$ such that:
either(i) $v(\gamma)$ is of the form $sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_2)$ and
(ii) $sat(s, \psi_2) \in lfp(T_{P_\mathcal{K},\beta 2})$
or there exists $s' \in D$ such that:
(i) $v(\gamma)$ is of the form:
$sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_1) \wedge t(s, s') \wedge sat(s', eu(\psi_1, \psi_2))$
(ii) $sat(s, \psi_1) \in lfp(T_{P_\mathcal{K},\beta 1})$,
(iii) $t(s, s') \in lfp(T_{P_\mathcal{K},0})$, and
(iv) $sat(s', eu(\psi_1, \psi_2)) \in lfp(T_{P_\mathcal{K},\alpha})$
(by the definitions of the encoding program $P_\mathcal{K}$ and $T_{P_\mathcal{K},0}$,
and recalling that $\sigma(t(s, s')) = 0$)

implies $sat(s, eu(\psi_1, \psi_2)) \in lfp(T_{P_\mathcal{K},\alpha})$
where $\alpha = \beta 1 + \beta 2 + 1 = size(eu(\psi_1, \psi_2))$
(by the definition of $T_{P_\mathcal{K},\alpha}$).

Case 6. (φ is $AF(\psi)$) Given a set of states Y, let $\tau_{AF}(Y)$ denote the set $[\psi] \cup \{s \in D \mid \text{for all } s' \in D \text{ if } s\,R\,s' \text{ then } s' \in Y\}$. From [21] we have that $\mathcal{K}, s \models AF(\psi)$ holds iff $s \in lfp(\tau_{AF})$. Thus, we have to show that, for all states $s \in D$, $s \in lfp(\tau_{AF})$ iff $sat(s, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$, where $\alpha = size(af(\psi))$. As in Case 5, we split the proof into two parts and we prove: (6.1) for all $h \geq 0$ and for all $s \in D$, $s \in \tau^h_{AF}(\emptyset)$ implies $sat(s, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$ and (6.2) for all $h \geq 0$ and for all $s \in D$, $sat(s, af(\psi)) \in T^h_{P_\mathcal{K},\alpha}(\emptyset)$ implies $s \in lfp(\tau_{AF})$.

We show the proof of (6.1) only. The proof of (6.2) is similar and is omitted.

We proceed by induction on h. The base case trivially holds because $\tau^0_{AF}(\emptyset) = \emptyset$. Now, we assume the following inductive hypothesis:

for all $s \in D$, $s \in \tau^h_{AF}(\emptyset)$ implies $sat(s, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$ (IndHyp3)

where $\alpha = size(eu(\psi_1, \psi_2))$, and we prove that, for all $s \in D$, $s \in \tau^{h+1}_{AF}(\emptyset)$ implies $sat(s, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$.
We have that:

$s \in \tau^{h+1}_{AF}(\emptyset)$

implies *either* $\mathcal{K}, s \models \psi$
or for all $s' \in D$, if $s\,R\,s'$ then $s' \in \tau^h_{AF}(\emptyset)$
(by the definition of τ_{AF})

implies *either* $sat(s, \psi) \in lfp(T_{P_\mathcal{K},\beta})$, where $\beta = size(\psi)$

or for all $s' \in D$, if $\mathcal{D} \models t_1(s,s') \vee \ldots \vee t_k(s,s')$ then $sat(s', af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$,
(by the definition of R and the inductive hypotheses IndHyp1 and IndHyp3)

implies *either* $sat(s,\psi) \in lfp(T_{P_\mathcal{K},\beta})$
or there exists $i \in \{1,\ldots,n\}$ such that:
for all $s' \in D$, if $\mathcal{D} \models cond_i(s) \wedge (act_{i1}(s,s') \vee \ldots \vee act_{iq_i}(s,s'))$
then $sat(s', af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$
where $cond_i(X) \wedge (act_{i1}(X,Y) \vee \ldots \vee act_{iq_i}(X,Y))$ is a nondeterministic event (by Proposition 3.1)

implies *either* $sat(s,\psi) \in lfp(T_{P_\mathcal{K},\beta})$
or there exist $i \in \{1,\ldots,n\}$ and $s_{i1},\ldots,s_{iq_i} \in D$ such that:
$\mathcal{D} \models cond_i(s) \wedge act_{i1}(s,s_{i1}) \wedge \ldots \wedge act_{iq_i}(s,s_{iq_i})$ and
$sat(s_{i1}, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$ and ... and $sat(s_{iq_i}, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$
(by Point 2 of Definition 4)

implies that there exist a valuation v and a clause $\gamma \in P_\mathcal{K}$ such that:
either(i) $v(\gamma)$ is of the form $sat(s, af(\psi)) \leftarrow sat(s,\psi)$ and
(ii) $sat(s,\psi) \in lfp(T_{P_\mathcal{K},\beta})$
or there exist $s_{i1} \in D,\ldots,s_{iq_i} \in D$ such that:
(i) $v(\gamma)$ is of the form:
$sat(s, af(\psi)) \leftarrow ts(s,[s_{i1},\ldots,s_{iq_i}]) \wedge sat\_all([s_{i1},\ldots,s_{iq_i}], af(\psi))$,
(ii) $ts(s,[s_{i1},\ldots,s_{iq_i}]) \in lfp(T_{P_\mathcal{K},0})$, and
(iii) $sat\_all([s_{i1},\ldots,s_{iq_i}], af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$
(by the definition of the encoding program $P_\mathcal{K}$ and $T_{P_\mathcal{K},0}$,
and recalling that $\sigma(ts(s,[s_{i1},\ldots,s_{iq_i}])) = 0$)

implies $sat(s, af(\psi)) \in lfp(T_{P_\mathcal{K},\alpha})$, where $\alpha = \beta + 1 = size(af(\psi))$
(by the definition of $T_{P_\mathcal{K},\alpha}$). □

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, "Regular model checking without transducers (On efficient verification of parameterized systems)," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Lecture Notes in Computer Science 4424, pp. 721–736, Springer-Verlag, 2007.

[3] G. R. Andrews, *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[4] K. R. Apt, "Introduction to logic programming," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 493–576, Elsevier, 1990.

[5] K. R. Apt, *Principles of Constraint Programming.* Cambridge University Press, 2003.

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

[7] T. Araki and T. Kasami, "Some decision problems related to the reachability problem for Petri nets," *Theoretical Computer Science*, vol. 3, no. 1, pp. 85–104, 1976.

[8] N. Bensaou and I. Guessarian, "Transforming constraint logic programs," *Theoretical Computer Science*, vol. 206, pp. 81–125, 1998.

[9] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, pp. 44–67, January 1977.

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

[11] 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.

[12] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.

[13] 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)*, pp. 238–252, ACM Press, 1977.

[14] 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.

[15] B. Cui and D. S. Warren, "A system for tabled constraint logic programming," 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. 478–492, Springer-Verlag, 2000.

[16] 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.

[17] G. Delzanno, "Verification of consistency protocols via infinite-state symbolic model checking," in *Proceedings of FORTE 2000*, vol. 183 of *IFIP Conference Proceedings*, pp. 171–186, Kluwer, 2000.

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

[19] 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.

[20] N. Dershowitz, "Termination of rewriting," *Journal of Symbolic Computation*, vol. 3, no. 1-2, pp. 69–116, 1987.

[21] E. A. Emerson and E. M. Clarke, "Characterizing correctness properties of parallel programs as fixpoints," in *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming (ICALP '81)*, Lecture Notes in Computer Science 85, (Berlin), pp. 169–181, Springer-Verlag, 1981.

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

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

[24] M. Falaschi and A. Villanueva, "Automatic verification of timed concurrent constraint programs," *Theory and Practice of Logic Programmimg*, vol. 6, no. 3, pp. 265–300, 2006. In: G. Delzanno, S. Etalle, and M. Gabbrielli (Eds.), Special Issue on Specification, Analysis, and Verification of Reactive Systems.

[25] F. Fioravanti, A. Pettorossi, and M. Proietti, "Automated strategies for specializing constraint logic programs," in *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '00), London, UK, 24-28 July, 2000* (K.-K. Lau, ed.), Lecture Notes in Computer Science 2042, pp. 125–146, Springer-Verlag, 2001.

[26] 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.

[27] L. Fribourg and H. Olsén, "A decompositional approach for computing least fixed-points of Datalog programs with Z-counters," *Constraints*, vol. 2, no. 3/4, pp. 305–335, 1997.

[28] 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.

[29] 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.

[30] J. Handy, *The Cache Memory Book*. Morgan Kaufman, 1998. Second Edition.

[31] 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.

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

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

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

[35] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] J. W. Lloyd, *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1987. Second Edition.

[42] M. J. Maher, "A transformation system for deductive database modules with perfect model semantics," *Theoretical Computer Science*, vol. 110, pp. 377–403, 1993.

[43] Z. Manna and A. Pnueli, "Models for reactivity," *Acta Informatica*, vol. 30, pp. 609–678, 1993.

[44] MAP, "The MAP transformation system." Available from `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2007.

[45] K. Marriott and P. Stuckey, *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[46] B. Martens and J. P. Gallagher, "Ensuring global termination of partial deduction while allowing flexible polyvariance," in *Proceedings of the 12th International Conference on Logic Programming (ICLP '95), June 13-16, 1995, Tokyo, Japan* (L. Sterling, ed.), pp. 597–611, The MIT Press, 1995.

[47] R. Mayr, "Decidability of model checking with the temporal logic EF," *Theoretical Computer Science*, vol. 256, no. 1-2, pp. 31–62, 2001.

[48] K. L. McMillan, S. Qadeer, and J. B. Saxe, "Induction in compositional model checking," in *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, Lecture Notes in Computer Science 1855, pp. 312–327, Springer, 2000.

[49] 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.

[50] A. Pettorossi and M. Proietti, "Synthesis and transformation of logic programs using unfold/fold proofs," *Journal of Logic Programming*, vol. 41, no. 2&3, pp. 197–230, 1999.

[51] A. Pettorossi and M. Proietti, "Perfect model checking via unfold/fold transformations," 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. 613–628, Springer-Verlag, 2000.

[52] 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.

[53] S. Prestwich, "Online partial deduction of large programs," in *Proceedings of the 1993 ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pp. 111–118, ACM Press, 1993.

[54] T. C. Przymusinski, "On the declarative semantics of stratified deductive databases and logic programs," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), pp. 193–216, Morgan Kaufmann, 1988.

[55] 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.

[56] W. Reisig, *Petri Nets - An Introduction.* Springer-Verlag, 1982.

[57] 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.

[58] A. Roychoudhury and C. R. Ramakrishnan, "Unfold/fold transformations for automated verification," in *Program Development in Computational Logic* (M. Bruynooghe and K.-K. Lau, eds.), Lecture Notes in Computer Science 3049, pp. 261–290, Springer, 2004.

[59] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson, "The XSB System, Version 2.2.," 2000.

[60] D. Sahlin, "Mixtus: An automatic partial evaluator for full Prolog," *New Generation Computing*, vol. 12, pp. 7–51, 1993.

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

[62] A. U. Shankar, "An introduction to assertional reasoning for concurrent systems," *ACM Computing Surveys*, vol. 25, pp. 225–262, Sept. 1993.

[63] N. Shankar, "Combining theorem proving and model checking through symbolic analysis," in *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, Lecture Notes in Computer Science 1877, pp. 1–16, Springer-Verlag, 2000.

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

[65] 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.

[66] H. Tamaki and T. Sato, "Unfold/fold transformation of logic programs," in *Proceedings of the Second International Conference on Logic Programming* (S.-Å. Tärnlund, ed.), (Uppsala, Sweden), pp. 127–138, Uppsala University, 1984.

[67] W. Thomas, "Languages, automata, and logic," in *Handbook of Formal Languages* (G. Rozenberg and A. Salomaa, eds.), vol. 3, pp. 389–455, Berlin: Springer, 1997.

[68] L. D. Zuck and A. Pnueli, "Model checking and abstraction to the aid of parameterized systems (A survey).," *Computer Languages, Systems & Structures*, vol. 30, no. 3-4, pp. 139–169, 2004.