ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

F. Fioravanti, A. Pettorossi, M. Proietti

VERIFYING CTL PROPERTIES OF INFINITE STATE SYSTEMS BY SPECIALIZING CONSTRAINT LOGIC PROGRAMS

R. 544 Gennaio 2001

Fabio Fioravanti – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email : `fioravan@iasi.rm.cnr.it`.
URL : `http://www.iasi.rm.cnr.it/~fioravanti`.

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

The first author has been partially supported by the Harmonise project of the European Community. This work is an extended version of the paper which appears in: M. Leuschel, A. Podelski, C.R. Ramakrishnan, U. Ultes-Nitsche (editors): Proceedings of the Second ACM-Sigplan International Workshop on Verification and Computational Logic, VCL'2001, Florence, Italy, September 4, 2001, pp. 85-96.

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. Our method makes use of constraint logic programs (CLP) with locally stratified negation and program specialization. We specify the temporal properties of a reactive system by means of a CLP program whose perfect model consists of all atoms $sat(s, \varphi)$ such that state s satisfies property φ. We verify a temporal property of a system by using a method based on the specialization of the corresponding CLP program w.r.t. the initial states of the system and the given property. Our specialization method makes use of: (i) a set of specialization rules which preserve the perfect model of CLP programs, and (ii) an automatic strategy which guides the application of the rules for making a proof of the property of interest. Our strategy always terminates and it is sound for the verification of properties which are expressible in the CTL branching time logic. Due to the undecidability of CTL for infinite state systems, our strategy is incomplete. However, by using a prototype implementation, we show that it is successful for the verification of several infinite state systems.

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

1. Introduction

Model checking is a highly successful technique for the automatic verification of properties of finite state reactive systems [9]. In essence, it consists in: (i) modeling the reactive system as a binary transition relation formalized as a *Kripke structure* over a *finite* set of states, (ii) expressing the property to be verified as a propositional temporal formula φ, and (iii) checking the satisfaction relation $\mathcal{K}, s \models \varphi$, where s is an initial state of the system, that is, checking that the formula φ holds in the state s of the structure $\mathcal{K}$.

The relation $\mathcal{K}, s \models \varphi$ is decidable for various classes of formulas and, in particular, there are very efficient algorithms for the case of formulas of the Computation Tree Logic (CTL, for short). CTL is a very expressive branching time temporal logic, where one may describe, among others, the so-called *safety* and *liveness* properties of reactive systems.

One of the most challenging problems in the area of verification of reactive systems, is how to extend model checking to *infinite* state reactive systems (see, for instance, [29]). In this case, a reactive system is modeled by an Kripke structure whose transition relation is over an infinite set of states. Several difficulties arise when considering model checking of infinite state systems and, in particular, in that case for most classes of formulas the satisfaction relation $\mathcal{K}, s \models \varphi$ is undecidable, and not even semidecidable.

In recent work three main approaches have been followed for dealing with this undecidability limitation.

The first approach consists in considering *decidable subclasses* of systems and formulas (see, for instance, [1, 13, 30]). By following this approach one may provide fully automatic techniques, which 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, [31, 39, 40]). This approach provides a great generality, but it needs some degree of human guidance, and this guidance may be difficult to provide when dealing with large systems.

The third approach consists in designing methods based on *abstractions*, that is, mappings for reducing an infinite state system (or a large finite state system) to a finite state one such that the properties of interest are preserved (see, for instance, [8, 10]). The choice of the suitable abstraction is crucial for the success of this kind of techniques. Once the abstraction is given, these techniques are fully automatic.

We propose a verification method which combines the generality of the approaches based on deduction with the mechanizability of the approaches based on abstractions. Our method is automatic, but incomplete, and its novelty resides in the idea of using: (1) *constraint logic programs* [20] (CLP, for short) for specifying reactive systems and their properties, and (2) *program specialization* [17, 19, 22, 24] as an inference mechanism for checking the properties of interest.

In our method, the transition relation which models the system of interest is specified by a finite collection of constraints over the infinite set of states. For any state s and CTL formula φ, the satisfaction relation $\mathcal{K}, s \models \varphi$ is encoded as a CLP program $P_{\mathcal{K}}$ which defines a binary predicate $sat(s, \varphi)$. For encoding negated CTL formulas, the program $P_{\mathcal{K}}$ uses *locally stratified* negation. The semantics of $P_{\mathcal{K}}$ is given by the *perfect model* $M(P_{\mathcal{K}})$, which is equal to the unique stable model and the two-valued, well-founded model [4]. Thus, we may check that $\mathcal{K}, s \models \varphi$ holds by checking that $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$.

In order to check whether or not $sat(s, \varphi)$ belongs to $M(P_{\mathcal{K}})$ for all initial states s, we propose a method based on the *specialization* of CLP programs. Program specialization is a transformation technique whose objective is the adaptation of a program to the context of use. In the case of CLP, program specialization can be defined as follows. We are given a program P and a goal of the form $c(X), p(X)$, where $c(X)$ is a constraint and $p(X)$ is an atom defined by P. We introduce the clause δ: $p_{spec}(X) \leftarrow c(X), p(X)$, where p_{spec} is a new predicate, and we want to derive a new program P_{spec} such that, for all ground terms d, if the constraint $c(d)$ holds then

$$p_{spec}(d) \in M(P \cup \{\delta\}) \quad \text{iff} \quad p_{spec}(d) \in M(P_{spec}) \qquad (\dagger)$$

We also want that checking whether or not $p_{spec}(d) \in M(P_{spec})$ be more efficient than checking whether or not $p_{spec}(d) \in M(P \cup \{\delta\})$.

Our verification method uses program specialization as follows. We consider the program $P_{\mathcal{K}}$ and we introduce the clause δ_{in}: $sat_{spec}(X) \leftarrow init(X),\ sat(X, \varphi)$, where sat_{spec} is a new predicate and $init(X)$ is a constraint which characterizes the initial states of the system, that is, $init(s)$ holds iff s is an initial state. By program specialization, from $P_{\mathcal{K}} \cup \{\delta_{in}\}$ we want to derive a new program $P_{\mathcal{K},spec}$ which contains the clause σ: $sat_{spec}(X) \leftarrow$. Indeed, by the equivalence ($\dagger$), if $\sigma \in P_{\mathcal{K},spec}$ then, for all initial states s, we have that $sat(s, \varphi) \in M(P_{\mathcal{K}})$ (see Section 6).

The specialization technique we use for program verification follows an approach based on transformation rules and strategies [7]. The transformation rules, which we will present below, are variants of the familiar unfolding, folding, clause deletion, and constraint replacement rules, and we show that they preserve the perfect model semantics, and thus, they ensure that the equivalence ($\dagger$) holds. We will also present a transformation strategy which guides the application of the transformation rules with the aim of deriving the clause $sat_{spec}(X) \leftarrow$. Our strategy is fully automatic and it always terminates. However, due to the above mentioned undecidability limitation, our strategy is incomplete, in the sense that it may be the case that $sat(s, \varphi) \in M(P_{\mathcal{K}})$ for all initial states s, and yet, our strategy terminates with a program $P_{\mathcal{K},spec}$ which does not contain the clause $sat_{spec}(X) \leftarrow$.

In order to ensure termination, our strategy uses a *generalization* technique which plays a role similar to that of abstraction in other verification methods such as [8, 10]. However, since generalization is applied during, and not before, the verification process, generalization may be more flexible than abstraction.

The contributions of this paper are the following ones. (i) We have shown that the CTL properties of *concurrent systems* as defined in [38], can be expressed by using perfect models of locally stratified CLP programs. (ii) We have defined variants of the usual transformation rules, such as, unfolding, folding, clause deletion, and constraint replacement. These variants 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 which makes program specialization always terminating. (iv) Finally, we have demonstrated that our technique is powerful enough to automatically verify several infinite state systems considered in the literature.

The structure of our paper is as follows. In Section 2 we present an introductory example to illustrate the basic ideas of our verification method. In Section 3 we recall some preliminary notions concerning locally stratified constraint logic programs and the CTL temporal logic. In Section 4 we consider a class of reactive systems and we show how CTL properties of systems in that class can be encoded by using locally stratified CLP programs. In Section 5 we present the transformation rules which we use for program specialization and we prove their correctness w.r.t. the perfect model semantics. In Section 6 we describe our strategy for program specialization, and we describe the technique for performing generalizations and ensuring the termination of the strategy. In Section 7 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 [15]. In particular, we have proved safety and liveness properties of the Bakery protocol and the Ticket protocol for mutual exclusion We have also proved a safety property of the Bounded Buffer protocol for ensuring no loss of messages. Finally, in Section 8 we compare our work with other verification techniques proposed in the literature. Among them we have given special attention to those techniques which use logic programming, constraints, tabled resolution, program analysis, and program transformation [11, 18, 26, 34, 35].

2. A Preliminary Example

In this section we illustrate by means of a simple example the basic ideas of our verification method. Let us consider a system *Count* 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.

Our verification method starts off by: (i) expressing the property of interest as a CTL formula φ, and (ii) providing a CLP program P_{Count} for the binary predicate *sat* such that φ holds in state X iff $sat(X,\varphi)$ belongs to the perfect model of P_{Count}. This can be done by using the algorithm we will give in Section 4. By doing so, we get for the system *Count*: (i) the CTL formula $\neg EF\ null$, where *null* is a property which holds in a state X iff $X=0$, and (ii) the following CLP program P_{Count}:

1. $sat(X, null) \leftarrow X=0$
2. $sat(X, \neg\varphi) \leftarrow \neg sat(X,\varphi)$
3. $sat(X, EF\ \varphi) \leftarrow sat(X,\varphi)$
4. $sat(X, EF\ \varphi) \leftarrow Y=X+1, sat(Y, EF\ \varphi)$

As indicated in Section 3.3, $\neg EF\ null$ expresses the fact that it is impossible to reach a state where *null* holds, and this property can be shown to hold in the initial state where $X=1$, by proving that $sat(1, \neg EF\ null) \in M(P_{Count})$.

Before making that proof, let us notice that by using SLDNF-resolution, the program P_{Count} does not terminate for the goal $sat(1, \neg EF\ null)$, because clause 4 allows us to get an infinitely failed SLDNF-tree containing the following infinite sequence of atoms:

$sat(1, EF\ null),\ sat(2, EF\ null),\ sat(3, EF\ null), \ldots$

Also by using tabled resolution [36], program P_{Count} fails to terminate because in the above sequence no atom is an instance of a preceding one.

Now we present the proof that $sat(1, \neg EF\ null) \in M(P_{Count})$ by using our verification method based on program specialization. We make use of the transformation rules which we will introduce in Section 5. These transformation rules are applied in an automatic way following the specialization strategy described in Section 6. This strategy starts off by introducing the definition (see rule R1):

5. $sat_{spec}(X) \leftarrow X=1,\ sat(X, \neg EF\ null)$

Then we unfold clause 5 (see rule R2p) and we get:

6. $sat_{spec}(X) \leftarrow X=1,\ \neg sat(X, EF\ null)$

Now we introduce the following new definition:

7. $newsat1(X) \leftarrow X=1,\ sat(X, EF\ null)$

and we fold clause 6 (see rule R3n) thereby deriving the clause:

8. $sat_{spec}(X) \leftarrow X=1,\ \neg newsat1(X)$

The specialization process continues by considering the new definition clause 7 and performing a sequence of transformation steps similar to the one performed starting from clause 5. By unfolding clause 7 we get:

9. $newsat1(X) \leftarrow X=1,\ X=0$
10. $newsat1(X) \leftarrow X=1,\ Y=X+1,\ sat(Y, EF\ null)$

Clause 9 is deleted because its body contains an unsatisfiable constraint (see rule R4f). In order to fold clause 10 we *generalize* the constraint $X=1, Y=X+1$ to the constraint $Y>1$ and we introduce the following new definition:

11. $newsat2(X) \leftarrow X>1,\ sat(X, EF\ null)$

whose body is obtained from the body of clause 10 by replacing $X=1, Y=X+1$ by $Y>1$ and applying a variable renaming. We fold clause 10 using clause 11 (see rule R3p) and we get:

12. $newsat1(X) \leftarrow X=1,\ Y=X+1,\ newsat2(Y)$

Now we consider the new definition clause 11 and, similarly to the two derivations which start from clauses 5 and 7, respectively, we perform unfolding, clause deletion, and folding steps as follows. We first unfold clause 11 and then apply the clause deletion rule, thereby deriving the following clause:

13. $newsat2(X) \leftarrow X>1,\ Y=X+1,\ sat(Y, EF\ null)$

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

14. $newsat2(X) \leftarrow X>1,\ Y=X+1,\ newsat2(Y)$

This folding step concludes the first phase of our specialization strategy (see Phase A of the strategy described in Section 6). At the end of this phase we have derived the following program:

8. $sat_{spec}(X) \leftarrow X=1,\ \neg newsat1(X)$
12. $newsat1(X) \leftarrow X=1,\ Y=X+1,\ newsat2(Y)$
14. $newsat2(X) \leftarrow X>1,\ Y=X+1,\ newsat2(Y)$

Now, since the bodies of the clauses which define the predicates $newsat1$ and $newsat2$, that is, clauses 12 and 14, have calls of $newsat2$, we deduce that for all integers n, all atoms of the form $newsat1(n)$ or $newsat2(n)$ are false in the perfect model of the program. Thus, we delete clauses 12 and 14 (by using rule R4u), and the literal $\neg newsat1(X)$ (by using rule R2n) in the body of clause 8 and we derive a specialized program P_{Spec} consisting of the following clause only:

15. $sat_{spec}(X) \leftarrow X=1$

Since we want to verify that $\neg EF\ null$ holds in the initial state, where the constraint $X=1$ is true, we may replace clause 15 (by using rule R5) by the following clause:

16. $sat_{spec}(X) \leftarrow$

Since clause 16 belongs to P_{spec}, we have that $sat_{spec}(1) \in M(P_{spec})$ and thus, by the correctness of program specialization (see Property (†) in Section 1), we also have that $sat_{spec}(1) \in M(P_{Count} \cup \{\text{clause}\, 5\})$. Since $M(P_{Count} \cup \{\text{clause}\, 5\})$ is a model of the completion of $P_{Count} \cup$

{clause 5} [4] and sat_{spec} is defined by clause 5 only, we get that $sat(1, \neg EF\ null) \in M(P_{Count})$ and this concludes our proof.

Before ending this section we want to briefly discuss the following points related to the proof we have done.
(1) The generation of a recurrent goal (in our case, the goal $X>1$, $sat(X, EF\ null)$ which occurs both in clause 11 and clause 13) during the unfolding process, determines after folding, the generation of recursive clauses (in our case, clause 14), and these recursive clauses allow us to infer that some atoms (in our case, $newsat2(X)$) are false in the perfect model of the program because they are infinitely failing.
(2) In order to perform the folding steps required for generating recursive clauses as indicated in Point (1) above, 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 10, and indeed, by using clause 11 we were able to fold all *sat* atoms occurring in the program at hand.
(3) The choice of a suitable generalization technique plays a crucial role in our verification method. Indeed, generalizations ensure termination of the specialization strategy, as it has been the case for our proof above, but they can also prevent the proof of the property of interest as we now indicate.

Indeed, if we had generalized the constraint $X=1$, $Y=X+1$ in the body of clause 10 to *true*, instead of $Y>1$, then, instead of clause 11, we would have introduced the following clause:

11*. $newsat2(X) \leftarrow sat(X, EF\ null)$

By unfolding clause 11* we would have derived the clause $newsat2(1) \leftarrow$ and we could have not inferred that for all integers n, $newsat2(n)$ is false in the perfect model of the program. As already mentioned, we will describe our generalization technique in Section 6.

3. Preliminaries

In this section we recall some basic notions of constraint logic programming. For notions not defined here the reader may refer to [3, 4, 20, 21, 27]. We also present our notational conventions and we briefly recall the syntax and the semantics of the Computational 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 [9].

3.1. 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 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 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 formulas by means of connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$) and quantifiers ($\exists$, $\forall$). As customary in logic programming, we may use comma, instead of $\wedge$.

Given a term or a formula e, 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* iff it contains no variable. Given a set $X = \{X_1, \ldots, X_n\}$ of n variables, by $\forall X\, \varphi$

we denote the formula $\forall X_1 \ldots X_n\, \varphi$. 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. By $\varphi(X_1, \ldots, X_n)$ we denote a formula whose free variables are among $X_1, \ldots, X_n$.

A *primitive constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol in $Pred_c$. The set $\mathcal{C}$ of *constraints* is the smallest set of formulas of $\mathcal{L}$ which contains all primitive constraints and it is closed w.r.t. conjunction and existential quantification. A *basic constraint* is either a primitive constraint or an existentially quantified constraint.

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. 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 $H \leftarrow c, G$, where: (i) H is an atom, called the *head* of γ and denoted $hd(\gamma)$, and (ii) c, G is a constrained goal, called the *body* of γ and denoted $bd(\gamma)$. 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 clause is *constraint-free* iff no constraints occur in its body.

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 $p(t_1, \ldots, t_n)$ and $p(u_1, \ldots, u_n)$, we denote by $p(t_1, \ldots, t_n) = p(u_1, \ldots, u_n)$ the conjunction of the constraints: $t_1 = u_1, \ldots, t_n = u_n$. We say that a term t is free for a variable X in a formula φ iff by substituting t for all free occurrences of X in φ, we do not introduce new occurrences of bound variables. A formula ψ is an *instance* of a formula φ iff 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.

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 $hd(\gamma)$. We say that the atom $p(t_1, \ldots, t_n)$ is *failed* in a program P iff $Def(p, P) = \emptyset$. We say that the atom $p(t_1, \ldots, t_n)$ is *valid* in a program P iff the fact $p(X_1, \ldots, X_n) \leftarrow$ belongs to P.

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

$p \leftarrow q, r$

$q \leftarrow p, \neg s$

$r \leftarrow$

p and q are useless predicates, while r is not useless. A clause γ is *useless* iff the predicate of $hd(\gamma)$ is useless.

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 $\rho(\varphi)$, called a *variant* of φ, obtained by replacing each (bound or free) variable X in φ by the variable $\rho(X)$. A clause γ is said to be *renamed apart* iff all its (bound or free) variables do not occur elsewhere.

We will feel free to apply to clauses the following transformations which, as the reader may verify, preserve program semantics (see Section 3.2):

(1) application of variable renamings,

(2) reordering of the constraints and the literals in the body (we will usually move all constraints to the left and all literals to the right), and

(3) replacement of a clause of the form $H \leftarrow X = t,\, c,\, G$, where $X \notin vars(t)$, by the clause $(H \leftarrow c, G)\{X/t\}$, and vice versa.

3.2. Semantics of Constraint Logic Programs

Now we introduce the notions of local stratification and perfect model for constraint logic programs. These notions are an extension of the similar notions for logic programs [4, 33] and are parametric w.r.t. the interpretation of the constraints [20, 21].

A *constraint domain* $\mathcal{D}$ consists of: (1) A non-empty set D, called *carrier*. (2) An assignment of a function $f_{\mathcal{D}}$ from D^n to D to each n-ary function symbol f in *Funct*. (3) An assignment of a *relation* over D^n to each n-ary constraint predicate symbol in $Pred_c$. In particular, $\mathcal{D}$ assigns the whole carrier D to *true*, the empty set to *false*, and the identity over D to the binary equality symbol $=$.

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, for reasons of simplicity, we will identify the constraint domain $\mathcal{D}$ 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 usually done in the first order predicate calculus.

Now given a program P we want to define the concept of $ground(P)$. A *valuation* is a function v: $Vars \to D$. We extend the domain of v to terms, constraints, and literals. 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 $v(t) = f_{\mathcal{D}}(v(t_1), \ldots, \sigma(t_n))$. Given a constraint c, $v(c)$ is the constraint obtained by replacing each free variable $X \in FV(c)$ by the ground term $v(X)$. Notice that $v(c)$ is a closed 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 $\neg v(A)$. We define $ground(P)$ as the following set of ground clauses:

$$ground(P) = \{v(H) \leftarrow v(L_1), \ldots, v(L_m) \mid v \text{ is a valuation, } (H \leftarrow c, L_1, \ldots, L_m) \in P, \text{ and } \mathcal{D} \models v(c)\}$$

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

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

Given a $\mathcal{D}$-interpretation I and a constraint-free, ground clause γ: $H \leftarrow L_1, \ldots, L_m$, we say that γ *is true in* I, written $I \models \gamma$ iff one of the following holds: (i) $H \in I$, or (ii) there exists $i \in \{1, \ldots, m\}$ such that L_i is an atom and $L_i \notin I$, or (iii) there exists $i \in \{1, \ldots, m\}$ such that L_i is a negated atom $\neg A_i$ and $A_i \in I$.

A $\mathcal{D}$-interpretation M is a $\mathcal{D}$-*model* of a (finite or infinite) set S of constraint-free, ground clauses iff for each clause γ in S, we have that $M \models \gamma$. M is a $\mathcal{D}$-*model* of a CLP program P iff M is a $\mathcal{D}$-*model* of $ground(P)$. It can be shown that the every set of definite, constraint-free, ground clauses has a least $\mathcal{D}$-model (w.r.t. set inclusion) and, thus, every definite constraint logic program has a least $\mathcal{D}$-model [21].

Now we define locally stratified constraint logic programs. 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. A clause δ in P is *locally stratified* w.r.t. a local stratification σ iff for all clauses of the form $A \leftarrow L_1, \ldots, L_m$ in $ground(\{\delta\})$ we have that for all $i = 1, \ldots, m$, if L_i is an atom B then $\sigma(A) \geq \sigma(B)$, otherwise, if L_i is a negated atom $\neg B$, then $\sigma(A) > \sigma(B)$. Given a local stratification σ, we say that program P is *locally stratified w.r.t.* σ iff every clause of P is locally stratified w.r.t. σ. A program P is *locally stratified* iff there exists a local stratification σ such that P is *locally stratified w.r.t.* σ. We denote by P_α the set of clauses in $ground(P)$ whose head is in stratum α.

A *level mapping* is a function from the set of predicate symbols to the finite ordinals. Given a level mapping λ, we extend it to literals as follows: if L is an atom $p(\ldots)$ then $\lambda(L) = \lambda(p)$, and if L is a negated atom $\neg p(\ldots)$ then $\lambda(L) = \lambda(p)$. A clause γ of the form $H \leftarrow c, L_1, \ldots, L_m$ is *stratified* w.r.t. a level mapping λ iff for all $i = 1, \ldots, m$, if L_i is a positive literal then $\lambda(H) \geq \lambda(L_i)$ and, if L_i is a negative literal then $\lambda(H) > \lambda(L_i)$. A program P is *stratified* iff 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 programs, 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, $\lambda(hd(\alpha)) < \lambda(hd(\beta))$ iff there exist i, j such that: (a) $i < j$, (b) $\alpha \in S_i$, and (c) $\beta \in S_j$. $S_1, \ldots, S_k$ are called the *strata* of P. Note that, as a consequence of the definition, the strata of a program are pairwise disjoint and if a program is stratified then it is locally stratified.

Similarly to the case of logic programs [4, 33], we define the *perfect model* $M(P)$ of a locally stratified constraint logic program P as the $\mathcal{D}$-interpretation $\bigcup_{\alpha \in W} M_\alpha$, where for every ordinal α in W, the set M_α is constructed as follows:
(1) M_0 is the empty set,
(2) if $\alpha > 0$, M_α is the least $\mathcal{D}$-model of the set of definite, constraint-free, ground clauses derived from P_α as follows: (i) every literal $\neg A$ occurring in the body of a clause in P_α is deleted iff A is in stratum τ, with $\tau < \alpha$, and $A \notin M_\tau$, and (ii) every clause γ in P_α is deleted iff there exists a literal $\neg A$ in $bd(\gamma)$ such that A is in stratum τ, with $\tau < \alpha$, and $A \in M_\tau$.

Similarly to the case of logic programs [4, 33], we have the following result.

Theorem 3.1. *Every locally stratified constraint logic program has a unique perfect model.*

In this paper we do not specify any particular method for solving constraints in $\mathcal{C}$. We only assume that there exists a computable total function *solve*: $\mathcal{C} \times \mathcal{P}_{\mathit{fin}}(\mathit{Vars}) \rightarrow \mathcal{C}$, where $\mathcal{P}_{\mathit{fin}}(\mathit{Vars})$ 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 $\mathit{solve}(c_1, X) = c_2$ 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) $\mathit{solve}(c, \emptyset) = \mathit{true}$ iff c is *satisfiable*, that is, $\mathcal{D} \models \exists(c)$, and
(ii) $\mathit{solve}(c, \emptyset) = \mathit{false}$ 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 6. The assumption that the *solve* function is complete w.r.t. satisfiability guarantees that constraint satisfiability tests, which are required in our verification method, are decidable and indeed they can be performed by applying the *solve* function.

Finally, we assume that, for any constraints c_1 and c_2, $\mathcal{D} \models \forall(c_1 \rightarrow c_2)$ is decidable.

3.3. The Computational Tree Logic

The Computation Tree Logic (CTL, for short) is a temporal logic for expressing properties of the evolutions in time of reactive systems. These evolutions are called *computation paths*. CTL formulas are built from a given set *Elem* of *elementary properties* by using: (i) the following linear-time operators along a computation path: G ('always'), F ('sometimes'), X ('nexttime'), and U ('until'), and (ii) the quantifiers over computation paths: A ('for all paths') and E ('for some path'), as indicated by the following definition.

Definition 1. [CTL Formulas] *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.

The other combinations of temporal operators and quantifiers are assumed to be abbreviations:
$EF\,\varphi \equiv EU(true, \varphi)$
$EG\,\varphi \equiv \neg AF\,\neg\varphi$
$AX\,\varphi \equiv \neg EX\,\neg\varphi$
$AU(\varphi_1, \varphi_2) \equiv \neg EU(\neg\varphi_2, \neg\varphi_1 \wedge \neg\varphi_2) \wedge (\neg EG\,\neg\varphi_2)$
$AG\,\varphi \equiv \neg EF\,\neg\varphi$
where *true* is the elementary property which holds in every state.

The semantics of CTL formulas is given by using a *Kripke structure* 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 $\models$ for denoting the satisfaction relation in a Kripke structure and the use of the same symbol for providing the semantics of constraint logic programs (Section 3.2).

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 the set of* initial states,
3. *$R \subseteq S \times S$ is a total relation, that is, for every state $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$. R is called a* transition relation, *and*
4. *$L : S \to \mathcal{P}(Elem)$ is a function which assigns to each state $s \in S$ a subset $L(s)$ of Elem, that is, a set of elementary properties which hold in s.*
A computation path in $\mathcal{K}$ from a state s_0 is an infinite *sequence of states $s_0 s_1 \ldots$ such that $(s_i, s_{i+1}) \in R$ for every $i \geq 0$.*

Given a Kripke structure $\mathcal{K} = \langle S, I, R, L\rangle$, the relation $\mathcal{K}, s \models \varphi$ is inductively defined as follows:

$\mathcal{K}, s \models e$ iff e is an elementary property in $L(s)$
$\mathcal{K}, s \models \neg\varphi$ 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 $\mathcal{K}, s_n \models \varphi$.

Notice that in the definition of the relation $\mathcal{K}, s \models \varphi$, the set I of initial states is not used. However, I has been introduced because it is often the case that the system properties we want to express are properties of the initial states of the system.

4. Expressing CTL Properties by Locally Stratified CLP

In this section we present the class of reactive systems which can be verified by using our method. This class is very general, and includes the *concurrent systems* defined in [38]. But, unlike [38], in order to specify these systems and their temporal properties, we use constraint logic programs. In this respect our approach is similar to the one presented in [11]. However, we use CLP programs with negation and the perfect model semantics, while the authors of [11] consider definite CLP programs and express temporal properties by means of least and greatest fixpoints.

A reactive system is modeled by a Kripke structure $\mathcal{K}$ [9] based on a constraint domain $\mathcal{D}$ as indicated below. Then, starting from $\mathcal{K}$ we constructs a locally stratified CLP program $P_\mathcal{K}$ which encodes the temporal properties of the system. The program $P_\mathcal{K}$ defines a binary predicate *sat* such that, for all states s and CTL formulas φ, we have that $\mathcal{K}, s \models \varphi$ iff $sat(s, \varphi) \in M(P_\mathcal{K})$.
A Kripke structure $\mathcal{K} = \langle S, I, R, L \rangle$ based on the constraint domain $\mathcal{D}$, is specified as follows. (We borrow some of the terminology from [38].)

1. The set S of states is the (possibly infinite) carrier D of the constraint domain $\mathcal{D}$.

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

3. The transition relation R is specified by a finite disjunction $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$ of constraints, so that for all states s_1 and s_2 in S, we have that: $(s_1, s_2) \in R$ iff $\mathcal{D} \models t_1(s_1, s_2) \vee \ldots \vee t_k(s_1, s_2)$
   Each disjunct $t_i(X,Y)$, called an *event*, is a constraint of the form: $cond_i(X) \wedge act_i(X,Y)$ such that
   3.1 $\mathcal{D} \models \forall X\, (cond_i(X) \rightarrow \exists Y\, act_i(X,Y))$, and
   3.2 $\mathcal{D} \models \forall X, Y, Z\, (act_i(X,Y) \wedge act_i(X,Z) \rightarrow Y = Z)$,
   The constraint $cond_i(X)$ is called the *enabling condition* and the constraint $act_i(X,Y)$ is called the *action*. Condition (3.1) means that act_i is defined whenever the corresponding enabling condition holds, and Condition (3.2) means that act_i is a function of its first argument.

4. The function $L : S \rightarrow \mathcal{P}(Elem)$, where *Elem* is the set of elementary properties of $\mathcal{K}$, is specified by associating a constraint $c_e(X)$ with each elementary property e, so that for all states $s \in S$, we have that: $e \in L(s)$ iff $\mathcal{D} \models c_e(s)$.

The construction of the CLP programs corresponding to Kripke structures, can be performed by using the *Encoding Algorithm* we now present.

The Encoding Algorithm.
Input: a Kripke structure $\mathcal{K} = \langle S, I, R, L \rangle$ based on a constraint domain $\mathcal{D}$.
Output: a locally stratified CLP program $P_\mathcal{K}$ such that, for all states $s \in S$ and for all CTL formulas φ, $\mathcal{K}, s \models \varphi$ iff $sat(s, \varphi) \in M(P_\mathcal{K})$.

Let us assume that R be specified by the disjunct: $t_1(X,Y) \vee \ldots \vee t_k(X,Y)$. Then the construction of $P_\mathcal{K}$ is done by induction on the structure of φ as follows.

(φ is the elementary property e) We introduce the clause:

$sat(X, e) \leftarrow c_e(X)$

where $c_e(X)$ is the constraint associated with the elementary property e.

(φ is $\neg\psi$) We introduce the clause:

$sat(X, \neg\psi) \leftarrow \neg sat(X, \psi)$

(The symbol $\neg$ in the head is a function symbol, while in the body $\neg$ is the negation connective.)

(φ is $\psi_1 \wedge \psi_2$) We introduce the clause:

$sat(X, \psi_1 \wedge \psi_2) \leftarrow sat(X, \psi_1), sat(X, \psi_2)$

(The symbol $\wedge$ in the head is a function symbol.)

(φ is $EX\,\psi$) For every $i = 1, \ldots, n$, we introduce the clause:

$sat(X, EX\,\psi) \leftarrow t_i(X, Y),\ sat(Y, \psi)$

(φ is $EU(\psi_1, \psi_2)$) We introduce the clause:

$sat(X, EU(\psi_1, \psi_2)) \leftarrow sat(X, \psi_2)$

and, for every $i = 1, \ldots, n$, we introduce the clause:

$sat(X, EU(\psi_1, \psi_2)) \leftarrow t_i(X, Y),\ sat(X, \psi_1),\ sat(Y, EU(\psi_1, \psi_2))$

(φ is $AF\,\psi$) Let us consider the disjunction $t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ of events, where for every $i = 1, \ldots, k$, $t_i(X, Y)$ is $cond_i(X) \wedge act_i(X, Y)$. We first rewrite that disjunction as a new disjunction $r_1(X, Y) \vee \ldots \vee r_n(X, Y)$, such that:
(1) for $i = 1, \ldots, n$, $r_i(X, Y)$ is a formula of the form $cond_i(X) \wedge (act_{i1}(X, Y) \vee \ldots \vee act_{im}(X, Y))$, called a *nondeterministic event*, where for $j = 1, \ldots, m$, $cond_i(X) \wedge act_{ij}(X, Y)$ is an event,
(2) for any two distinct i and l in $\{1, \ldots, n\}$, $cond_i(X)$ and $cond_l(X)$ are mutually exclusive, that is, $\mathcal{D} \models \neg\exists X\ cond_i(X) \wedge cond_l(X)$, and
(3) $\mathcal{D} \models \forall X, Y\ ((t_1(X, Y) \vee \ldots \vee t_k(X, Y)) \leftrightarrow (r_1(X, Y) \vee \ldots \vee r_n(X, Y)))$.
We introduce the following clause:

$sat(X, AF\,\psi) \leftarrow sat(X, \psi)$

and, for $i = 1, \ldots, n$, we introduce the clause,

$$\begin{aligned} sat(X, AF\,\psi) \leftarrow\ & cond_i(X) \wedge act_{i1}(X, Y_{i1}) \wedge \ldots \wedge act_{im}(X, Y_{im}), \\ & sat(Y_{i1}, AF\,\psi), \ldots,\ sat(Y_{im}, AF\,\psi) \end{aligned}$$

where $X, Y_{i1}, \ldots, Y_{im}$ are distinct variables.

The rewriting needed for the case where φ is $AF\,\psi$ is always possible for Kripke structures based on a constraint domain $\mathcal{D}$ which satisfies the following property:

(*Property* P) For every constraint $c(X)$, the formula $\neg c(X)$ is equivalent to a *finite* disjunction $c_1(X) \vee \ldots \vee c_m(X)$ of pairwise mutually exclusive constraints.

This Property P can be formally expressed as follows: for every constraint $c(X)$ in $\mathcal{D}$ there exist the 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 l in $\{1, \ldots, m\}$, $c_i(X) \wedge c_l(X)$ is unsatisfiable, that is, $\mathcal{D} \models \neg\exists X\ (c_i(X) \wedge c_l(X))$.
If Property P holds we also 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 $\neg c(X)$.

Example 1. Let us consider the constraint domain $\mathcal{R}_{lin}$ of linear equations ($=$) and inequations ($<, \leq$) over real numbers. Without loss of generality, we may assume that every constraint in $\mathcal{R}_{lin}$ is a conjunction of constraints of the form $t_1\ op\ t_2$, where $op \in \{=, <, \leq\}$ and t_1 and t_2

are terms built out of reals, variables, and arithmetic operators. Then, the negation of any constraint in $\mathcal{R}_{lin}$ can be partitioned into a finite disjunction of constraints, because:

(i) $\mathcal{R}_{lin} \models \forall X\, (\neg\, t_1 = t_2 \;\leftrightarrow\; (t_1 < t_2 \vee t_2 < t_1))$

(ii) $\mathcal{R}_{lin} \models \forall X\, (\neg\, t_1 < t_2 \;\leftrightarrow\; t_2 \leq t_1)$.

However, if we consider the domain $\mathcal{FT}$ of equations between finite terms which are built out of an infinite set of function symbols, then in $\mathcal{FT}$ 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:

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

where G denotes the infinite set of all ground terms. If we consider the domain of equations between finite 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 the constraint $X = s(0)$ can be partitioned into $X = 0 \vee \exists Y\, X = s(s(Y))$.

The above Encoding Algorithm can easily be extended by considering the cases where the outermost operator of the formula φ is one of the following: *EF*, *EG*, *AX*, *AU*, and *AG*. In order to do so it is enough to express these operators in terms of *EX*, *EU*, and *AF*. For instance, if φ is $EF\,\psi$ we introduce the following clause:

$sat(X, EF\,\psi) \leftarrow sat(X, \psi)$,

and for $i = 1, \ldots, n$, we introduce the clause:

$sat(X, EF\,\psi) \leftarrow t_i(X, Y),\; sat(Y, EF\,\psi)$

because: (i) $EF\,\psi$ stands for $EU(true, \psi)$ and (ii) $sat(X, true)$ is true for all states X.

The program $P_{\mathcal{K}}$ constructed by the Encoding Algorithm is locally stratified w.r.t. the function σ defined as follows: for every $s \in S$ and for every CTL formulas φ, $\sigma(sat(s, \varphi)) = length(\varphi)$, where $length(\varphi)$ is the number of occurrences of elementary properties, connectives, and operators occurring in φ.

We have the following theorem.

Theorem 4.1. [Correctness of the Encoding Algorithm] *Let $\mathcal{K} = \langle S, I, R, L\rangle$ be a Kripke structure and let $P_{\mathcal{K}}$ be the locally stratified program constructed from $\mathcal{K}$ by the Encoding Algorithm. For all states $s \in S$ and CTL formulas φ, we have that: $\mathcal{K}, s \models \varphi$ iff $sat(s, \varphi) \in M(P_{\mathcal{K}})$.*

Proof. The proof is by structural induction on φ.

(φ is $e \in Elem$) For all states $s \in S$ we have that:

$\mathcal{K}, s \models e$ iff $\mathcal{D} \models c(s)$ (by the assumption on elementary properties)

iff $sat(s, e) \in M(P_{\mathcal{K}})$ (by the Encoding Algorithm).

(φ is $\neg\psi$) For all states $s \in S$ we have that:

$\mathcal{K}, s \models \neg\psi$ iff $\mathcal{K}, s \models \psi$ does not hold (by the semantics of CTL)

iff $sat(s, \psi) \notin M(P_{\mathcal{K}})$ (by induction hypothesis)

iff $sat(s, \neg\psi) \in M(P_{\mathcal{K}})$ (by the Encoding Algorithm).

(φ is $\psi_1 \wedge \psi_2$) For all states $s \in S$ 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 semantics of CTL)

iff $sat(s, \psi_1) \in M(P_{\mathcal{K}})$ and $sat(s, \psi_2) \in M(P_{\mathcal{K}})$ (by induction hypothesis)

iff $sat(s, \psi_1 \wedge \psi_2) \in M(P_{\mathcal{K}})$ (by the Encoding Algorithm).

(φ is $EX\,\psi$) For all states $s \in S$ we have that:
$\mathcal{K}, s \models EX\,\psi$ iff there exists a state $s_1 \in S$ such that $(s, s_1) \in R$ and $\mathcal{K}, s_1 \models \psi$ (by the semantics of CTL)
iff $\exists s_1 \in S$ and $\exists j \in \{1, \ldots, k\}$ such that $\mathcal{D} \models t_j(s, s_1)$ and $sat(s_1, \psi) \in M(P_\mathcal{K})$ (by the assumption on the transition relation and induction hypothesis)
iff $\exists s_1 \in S$ and there exists a clause $\gamma \in ground(P_\mathcal{K})$ of the form:
$sat(s, EX\,\psi) \leftarrow sat(s_1, \psi)$ and $sat(s_1, \psi) \in M(P_\mathcal{K})$ (by the Encoding Algorithm)
iff $sat(s, EX\,\psi) \in M(P_\mathcal{K})$ (by definition of $M(P_\mathcal{K})$).
In the rest of the proof: (i) *lfp* denotes the least fixpoint operator, and (ii) given a formula φ, we denote by $[\varphi]$ the set $\{s \in S \mid \mathcal{K}, s \models \varphi\}$, that is, the set of states in which φ is true.
(φ is $EU(\psi_1, \psi_2)$) From [12] we have that $\mathcal{K}, s \models EU\,(\psi_1, \psi_2)$ holds iff $s \in lfp(\tau_{EU})$, where $\tau_{EU} = \lambda I.[\psi_2] \cup ([\psi_1] \cap EX^{-1}(I))$, and $EX^{-1}(I) = \{s \in S \mid \exists s' \in I$ such that $(s, s') \in R\}$. Now let us consider the operator $T_{EU} : \mathcal{P}(S) \to \mathcal{P}(S)$ defined as follows:

$$\begin{aligned} T_{EU}(I) = \{s \in S \quad | \quad & sat(s, \psi_2) \in M(P_\mathcal{K}) \text{ or} \\ & sat(s, \psi_1) \in M(P_\mathcal{K}) \text{ and } \exists s' \in I \text{ such that } (s, s') \in R\} \end{aligned}$$

By structural induction we have that, for $i = 1, 2$, $\mathcal{K}, s \models \psi_i$ iff $sat(s, \psi_i) \in M(P_\mathcal{K})$ and, thus, we easily get that $s \in lfp(\tau_{EU})$ iff $s \in lfp(T_{EU})$. It remains to show that for all $s \in S$, $s \in lfp(T_{EU})$ iff $sat(s, EU(\psi_1, \psi_2)) \in M(P_\mathcal{K})$. This proof, which is left to the reader, is similar to the one of Theorem 6.5 [27, page 38], which states that the least Herbrand model of a definite logic program P is the least fixpoint of its T_P operator.
(φ is $AF\,\psi$) From [12] $\mathcal{K}, s \models AF\,\psi$ holds iff $s \in lfp(\tau_{AF})$, where $\tau_{AF} = \lambda I.[\psi] \cup AX^{-1}(I)$ and $AX^{-1}(I) = \{s \in S \mid \forall s' \in S$ if $(s, s') \in R$ then $s' \in I\}$. Now let us consider the operator $T_{AF} : \mathcal{P}(S) \to \mathcal{P}(S)$ defined as follows:

$$\begin{aligned} T_{AF}(I) = \{s \in S \quad | \quad & sat(s, \psi) \in M(P_\mathcal{K}) \text{ or} \\ & \forall s' \in S \text{ if } (s, s') \in R \text{ then } s' \in I\} \end{aligned}$$

By structural induction we have that $\mathcal{K}, s \models \psi$ iff $sat(s, \psi) \in M(P_\mathcal{K})$, and thus, we easily get that $s \in lfp(\tau_{AF})$ iff $s \in lfp(T_{AF})$. It remains to show that for all $s \in S$, $s \in lfp(T_{AF})$ iff $sat(s, AF\,\psi) \in M(P_\mathcal{K})$. Again, this proof is similar to the one of Theorem 6.5 [27, page 38] and we leave to the reader. Recall that, in this case, when writing the clauses for $AF\,\psi$ in $P_\mathcal{K}$, we assume that the relation R is specified by a disjunction of nondeterministic events as indicated in the Encoding Algorithm. □

In the following example we consider a simple reactive system modeled by a Kripke structure $\mathcal{K}$ and we apply our Encoding Algorithm for generating the corresponding program $P_\mathcal{K}$.

Example 2. 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 real number. The Kripke structure $\mathcal{K} = \langle S, I, R, L\rangle$ which models that system can be defined as follows.
$\mathcal{K}$ is based on the constraint domain $\mathcal{D}$ whose carrier is the set $S = \{a, b\} \times \mathbf{R}$, where $\mathbf{R}$ is the set of real numbers. In $\mathcal{D}$ we have: (i) the addition between real numbers, (ii) equations between elements in $\{a, b\}$, and (iii) equations and inequations between reals. For equations between elements in $\{a, b\}$ and equations between reals we use the same symbol $=$.

The set I of the initial states is specified by the constraint $init(X_1, X_2) \equiv X_1 = a,\ X_2 = 0$, that is, I is the singleton $\{\langle a, 0\rangle\}$. (Notice that to represent states we use two variables, instead of a single variable ranging over pairs.)

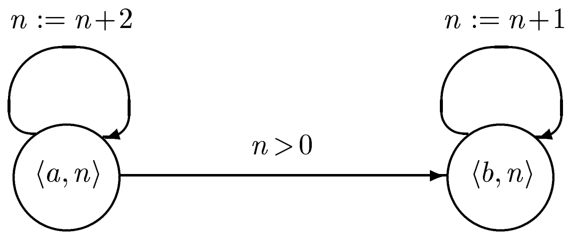

Figure 1: A simple reactive system.

The transition relation R is specified as the disjunction of the following three events:

$$
\begin{aligned}
t_1(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = a) \wedge (Y_1 = a \wedge Y_2 = X_2 + 2)\\
t_2(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = a \wedge X_2 > 0) \wedge (Y_1 = b \wedge Y_2 = X_2)\\
t_3(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = b) \wedge (Y_1 = b \wedge Y_2 = X_2 + 1)
\end{aligned}
$$

where in each disjunct the parentheses are used to distinguish between the enabling conditions and the actions.

We specify the elementary property *neg* which holds in a state $\langle X_1, X_2\rangle$ iff $X_2 < 0$. 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 $\langle a, 0\rangle$. In order to verify this property, we first apply the Encoding Algorithm thereby deriving the program $P_{\mathcal{K}}$ such that $\mathcal{K}, \langle a, 0\rangle \models \neg AF\, neg$ iff $sat(a, 0, \neg AF\, neg) \in M(P_{\mathcal{K}})$.

Notice that the conditions occurring in the events $t_1(X_1, X_2, Y_1, Y_2)$ and $t_2(X_1, X_2, Y_1, Y_2)$ 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 operator AF we have to perform the rewriting described in the Encoding Algorithm. This rewriting can indeed be performed because the constraint domain $\mathcal{D}$ satisfies Property P (see also Example 1). In particular, we use the following equivalences:

$$
\begin{aligned}
&\mathcal{D} \models \forall X((\neg\, X > 0) \leftrightarrow X \leq 0)\\
&\mathcal{D} \models \forall X((\neg\, X = a) \leftrightarrow X = b)
\end{aligned}
$$

Thus, we specify the transition relation by using the disjunction of the following three nondeterministic events:

$$
\begin{aligned}
r_1(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = a \wedge X_2 \leq 0) \wedge (Y_1 = a \wedge Y_2 = X_2 + 2)\\
r_2(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = a \wedge X_2 > 0) \wedge\\
&\quad\;((Y_1 = a \wedge Y_2 = X_2 + 2) \vee (Y_1 = b \wedge Y_2 = X_2))\\
r_3(X_1, X_2, Y_1, Y_2) &\equiv (X_1 = b) \wedge (Y_1 = b \wedge Y_2 = X_2 + 1)
\end{aligned}
$$

The application of the Encoding Algorithm produces a program $P_{\mathcal{K}}$ containing the following clauses (we do not list the clauses for the operators EX and EU because they are not needed for verifying our property $\neg AF\, neg$):

$$
\begin{aligned}
&sat(X_1, X_2, neg) \leftarrow X_2 < 0\\
&sat(X_1, X_2, \neg\varphi) \leftarrow \neg sat(X_1, X_2, \varphi)\\
&sat(X_1, X_2, AF\,\varphi) \leftarrow sat(X_1, X_2, \varphi)\\
&sat(X_1, X_2, AF\,\varphi) \leftarrow X_1 = a,\ X_2 \leq 0,\ X_3 = X_2 + 2,\ sat(X_1, X_3, AF\,\varphi)\\
&sat(X_1, X_2, AF\,\varphi) \leftarrow X_1 = a,\ X_2 > 0,\ X_3 = b,\ X_4 = X_2 + 2,\\
&\qquad\qquad\qquad\qquad\quad sat(X_1, X_4, AF\,\varphi),\ sat(X_3, X_2, AF\,\varphi)\\
&sat(X_1, X_2, AF\,\varphi) \leftarrow X_1 = b,\ X_3 = X_2 + 1,\ sat(X_1, X_3, AF\,\varphi)
\end{aligned}
$$

5. The Rules for Specializing CLP Programs

The process of specializing a given program P whereby deriving program Q, can be formalized as a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where $P_0 = P$, $P_n = Q$ and, 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 presented in this paper, of the rules considered in the literature for transforming logic programs and constraint logic programs (see, in particular, [5, 14, 17, 28, 37, 41]).

R1. Constrained Atomic Definition. We introduce the clause, called a *definition*,

$\delta: \; \mathit{newp}(X_1, \ldots, X_m) \leftarrow c, A$

where: (i) *newp* is a predicate symbol not occurring in $P_0, \ldots, P_k$, (ii) $\mathit{vars}(A) = \{X_1, \ldots, X_m\}$, (iii) $FV(c) \subseteq \{X_1, \ldots, X_m\}$, and (iv) the predicate of A occurs in P_0.
By *constrained atomic definition* (or *definition*, for short), we derive the new program $P_{k+1} = P_k \cup \{\delta\}$.
For $i \geq 0$, Defs_i is the set of definitions introduced during the transformation sequence $P_0, \ldots, P_i$. In particular, $\mathit{Defs}_0 = \emptyset$.

R2. Unfolding. Let $\gamma: \; H \leftarrow c, G_1, L, G_2$ be a renamed apart clause in P_k. We consider the following two cases.
(R2p: *Positive Unfolding*) Let L be an atom A. By unfolding γ w.r.t. A we derive the set of clauses

$$\Gamma: \; \{H \leftarrow c,\, A{=}K,\, d,\, G_1, B, G_2 \mid A \text{ and } K \text{ have the same predicate symbol and } K \leftarrow d, B \text{ is a clause in } P_k\}$$

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

(R2n: *Negative Unfolding*) Let L be a negated atom $\neg A$.
(i) If A is failed in P_k, then by unfolding γ w.r.t. $\neg A$ we derive the clause
$\eta: \; H \leftarrow c, G_1, G_2$
and we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.
(ii) If A is valid in P_k, then by unfolding γ w.r.t. $\neg A$ we derive the new program $P_{k+1} = P_k - \{\gamma\}$.

R3. Constrained Atomic Folding. Let $\gamma: \; H \leftarrow c, G_1, L, G_2$ be a clause in P_k, where L is either an atom A or a negated atom $\neg A$. Let $\delta: \; N \leftarrow d, B$ be an instance of a clause in Defs_k such that: (i) A and B have the same predicate symbol and (ii) $\mathcal{D} \models \forall\, (c \rightarrow (d \wedge A{=}B))$.

(R3p: *Positive Folding*) Let L be A. By folding γ w.r.t. L using δ we derive the clause

$\eta: \; H \leftarrow c, G_1, N, G_2$

and we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.
(R3n: *Negative Folding*) Let L be $\neg A$. By folding γ w.r.t. L using δ, we derive the clause

$\eta: \; H \leftarrow c, G_1, \neg N, G_2$

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

R4. Clause Removal. Let γ be a clause in P_k. We derive the new program $P_{k+1} = P_k - \{\gamma\}$ if one of the following cases occurs:

(R4f: *Unsatisfiable Body*) γ is the clause $H \leftarrow c, G$ and the constraint c is unsatisfiable, that is, $\mathcal{D} \models \neg\exists(c)$;

(R4s: *Subsumed Clause*) γ is the clause $H \leftarrow c, G$, with $(c, G) \neq \mathit{true}$, and H is valid in P_k;

(R4u: *Useless Clause*) γ is useless in P_k.

R5. Contextual Constraint Replacement. Let $\mathbf{C}$ be a set of constrained atoms. Let γ_1 be a renamed apart clause in P_k of the form: $H \leftarrow c_1, G$. Suppose that for some constraint c_2, and for every constrained atom c, A in $\mathbf{C}$, we have that:

$\mathcal{D} \models \forall((c \wedge A{=}H) \rightarrow (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2))$

where: (i) $Y{=}FV(c_1){-}vars(\{H,G\})$, and (ii) $Z{=}FV(c_2){-}vars(\{H,G\})$. Then, by contextual constraint replacement w.r.t. $\mathbf{C}$ we derive the clause

$\gamma_2: \;\; H \leftarrow c_2, G$

and we derive the new program $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$.
For $\mathbf{C} = \{(true, H)\}$ we have the following particular case of the contextual constraint replacement rule.
(R5r: *Constraint Replacement*) Let $\gamma_1 : \; H \leftarrow c_1, G$ 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) $Y = FV(c_1){-}vars(\{H,G\})$, and (ii) $Z = FV(c_2){-}vars(\{H,G\})$. Then we derive the clause

$\gamma_2: \;\; H \leftarrow c_2, G$

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

The following theorems ensure the correctness of the transformation rules w.r.t. the perfect model semantics. Proofs of these results may be found in [16].

Theorem 5.1. [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 during the construction of $P_0, \ldots, P_n$,*
(i) *each clause introduced by the constrained atomic definition rule and used for constrained atomic folding, is unfolded w.r.t. the atom in its body,*
(ii) *the contextual constraint replacement rule* R5 *is only applied in its restricted form* R5r.
Then P_n is locally stratified and $M(P_0 \cup Defs_n) = M(P_n)$.

In order to state the correctness of the contextual constraint replacement rule R5 w.r.t. the perfect model semantics we need the following definition.

Definition 3. [Call Patterns] *Let γ be a clause of the form $H \leftarrow c, L_1, \ldots, L_k$, where for $i = 1, \ldots, k$, L_i is either an atom A_i or a negated atom $\neg A_i$. For $i = 1, \ldots, k$, the constrained atom $(solve(c, X), A_i)$, where $X = vars(A)$, is said to be the* call pattern *of L_i in γ. By $CP(\gamma)$ we denote the set of all call patterns of literals in γ, that is, $\{(solve(c,X), A_i) \,|\, i = 1, \ldots, k\}$. Given a program P, we define $CP(P) = \bigcup_{\gamma \in P} CP(\gamma)$.*

Call patterns will be used in our contextual specialization strategy below (see Section 6) for introducing new definitions and for applying the contextual constraint replacement rule R5.

Theorem 5.2. [Correctness of the Contextual Constraint Replacement Rule] *Let $P_0, \ldots, P_n$ be a transformation sequence where, for $k = 0, \ldots, n-1$, program P_{k+1} is derived from P_k by applying the contextual constraint replacement rule* R5 *w.r.t. a set $\mathbf{C}$ of constrained atoms such that $\mathbf{C} \supseteq CP(P_0)$. Assume that P_0 and P_n are locally stratified w.r.t. the same local stratification function. Then for all constrained atoms $c, A \in \mathbf{C}$ and for every valuation v, we have that:*

if $\mathcal{D} \models v(c)$ then $\;\; v(A) \in M(P_0)$ iff $v(A) \in M(P_n)$

Notice that if P_n is derived from P_0 by applications of the contextual constraint replacement rule, then it may be the case that $M(P_0) \neq M(P_n)$, because Theorem 5.2 ensures the preservation of the perfect model only for atoms whose arguments satisfy the constraints specified by $\mathbf{C}$.

Notice also that, the contextual constraint replacement rule may not preserve local stratification. For instance, let us consider the program

P_0: $p \leftarrow \mathit{false}, \neg p$

We have that $\mathit{ground}(P_0) = \emptyset$, and thus, P_0 is locally stratified w.r.t. every local stratification function. Now, by applying the contextual constraint replacement rule w.r.t. $\{(\mathit{false}, p)\}$ we get the program

P_1: $p \leftarrow \neg p$

which is *not* locally stratified.

However, we will see in Section 6 that all applications of the contextual constraint replacement rule required by our verification technique, do preserve local stratification. This is due to the fact that, according to the specialization strategy presented in Section 6, we apply the contextual constraint replacement rule to *stratified* programs only, and the contextual constraint replacement rule preserves stratification. Indeed, let us consider a clause γ_1: $H \leftarrow c_1, G$. If γ_1 is stratified w.r.t. a level mapping λ, then also the clause $H \leftarrow c_2, G$, derived by replacing c_1 by c_2 in γ, is stratified w.r.t. λ because the user defined predicates occurring in γ_1 and γ_2 are the same. Thus, we have the following straightforward consequence of Theorem 5.2.

Corollary 5.3. [Correctness of the Contextual Constraint Replacement Rule for Stratified Programs] *Let P_0 be a stratified program and let $P_0, \ldots, P_n$ be a transformation sequence where, for $k = 0, \ldots, n-1$, program P_{k+1} is derived from P_k by applying the contextual constraint replacement rule* R5 *w.r.t. a set $\mathbf{C}$ of constrained atoms such that $\mathbf{C} \supseteq CP(P_0)$. Then* (i) *$P_n$ is stratified and* (ii) *for all constrained atoms $c, A \in \mathbf{C}$ and for every valuation v, we have that:*

if $\mathcal{D} \models v(c)$ *then* $\quad v(A) \in M(P_0)$ *iff* $v(A) \in M(P_n)$

Now we compare the transformation rules considered in this section with other sets of transformation rules which have been presented in the literature.

The rules listed above are an extension to locally stratified CLP programs of the rules presented in [17] for the case of definite CLP programs.

The rules of [5, 14] are extensions of Tamaki-Sato's unfold/fold rules [41] to the case of definite CLP programs. If we consider definite CLP programs only, the definition and folding rules of [5, 14] are more general than ours, because they allow the body of a definition to be a non-atomic constrained goal. However, in Section 7 we show that by applying our method, which employs atomic definitions and atomic folding only, we are able to verify several interesting CTL properties of infinite state systems.

The transformation rules considered by Seki [37] extend Tamaki-Sato's rules to the case of logic programs with negation, but without constraints. One further difference between our rules and those in [37] is that, Seki's folding rule can be used for replacing a clause γ: $H \leftarrow c, G_1, \neg A, G_2$ by a new clause γ_1: $H \leftarrow c, G_1, \mathit{newp}(\ldots), G_2$, but not by a new clause γ_2: $H \leftarrow c, G_1, \neg \mathit{newp}(\ldots), G_2$. The replacement of clause γ by clause γ_2 is possible by using our folding rule R3n.

For a similar reason, our folding rule differs from the folding rule for stratified constraint logic programs considered by Maher [28]. Moreover, Maher's folding rule does not permit the derivation of recursive clauses, and the derivation of recursive clauses is very important in our method (see Section 6).

Finally, the rule for deleting useless clauses and the contextual constraint replacement rule (except for the particular case of rule R5r) are novel and they are not present in [5, 14, 28, 37].

6. The Specialization Strategy

Now we present the specialization strategy which we use for verifying CTL properties of reactive systems.

Suppose that we are given a reactive system modeled 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, $\mathcal{K}, s \models \varphi$. By Theorem 4.1, in order to do this verification it is enough to verify that, for all initial states s, $sat(s, \varphi) \in M(P_{\mathcal{K}})$, where $P_{\mathcal{K}}$ is the locally stratified CLP program constructed according to our Encoding Algorithm.

We start off by introducing the clause δ_{in}: $sat_{spec}(X) \leftarrow init(X),\ sat(X, \varphi)$, where sat_{spec} is a new predicate and $init(X)$ is the constraint which specifies the initial states of the system (see Section 4). Then we apply the transformation rules of Section 5, according to the specialization strategy presented below, with the aim of deriving a program $P_{\mathcal{K},spec}$ containing the fact $sat_{spec}(X) \leftarrow$. If we succeeds in doing so, we have that for all states s, if $init(s)$ holds then $sat(s, \varphi) \in M(P_{\mathcal{K}})$ (see Theorem 6.2).

Our specialization strategy is divided into three phases, called *Phase A*, *B*, and *C*, respectively.

Phase A starts off by unfolding the definition δ and then applying the constraint replacement rule for simplifying the constraints as much as possible. By doing so, we derive a new set of clauses, say Γ. Then we apply a *generalization function* and we introduce a (possibly empty) set of new definitions of the form $newp(X) \leftarrow d(X), sat(X, \psi)$ such that we can apply the folding rule w.r.t. each constrained literal $sat(X, \psi)$ or $\neg sat(X, \psi)$ occurring in the body of a clause in Γ. We iterate unfolding, generalization, and folding steps, for each new definition introduced by generalization, and we stop this iteration when no new definitions are necessary for applying the folding rule w.r.t. all (positive or negative) occurrences of sat literals, because we can fold those occurrences by using definitions which have been already introduced.

Below we will present the generalization function which ensures that a finite set of definitions will be introduced and, thus, Phase A of the specialization strategy always terminates (see Theorem 6.3). At the end of Phase A we derive a program P_A where the (positive or negative) dependencies among sat atoms have been lifted to dependencies among newly introduced predicates. In particular, due to the structure of the CTL formulas which occur as second arguments of the predicate sat, we always derive a *stratified* program. This property will be exploited during Phase C.

In order to derive a program $P_{\mathcal{K},spec}$ which contains the fact $sat_{spec}(X) \leftarrow$, that is, a program where $sat_{spec}(X)$ is valid, we may need to derive programs where the atoms of the form $newp(X)$ are either valid or failed. This can be accomplished during Phases B and C of our specialization strategy by applying the following rules: (i) positive and negative unfolding, (ii) removal of useless and subsumed clauses, and (iii) constraint replacement, as the following example illustrates.

Example 3. Let us assume that the output of Phase A is the following program P_A:

1. $sat_{spec}(X) \leftarrow X=0,\ newsat1(X),\ \neg newsat2(X)$
2. $newsat1(X) \leftarrow X \geq 0$
3. $newsat1(X) \leftarrow X \geq 0,\ Y=X+1,\ newsat1(Y)$
4. $newsat2(X) \leftarrow X \geq 0,\ Y=X-1,\ newsat2(Y)$

Suppose that $init(X)$ is the constraint $X=0$ in clause 1. From P_A we want to derive a program containing the fact $sat_{spec}(X) \leftarrow$. In order to do so, we first derive a program where $newsat1(X)$ is valid and $newsat2(X)$ is failed. We proceed as follows. We notice that the constraint $X \geq 0$ in the body of clause 2 is redundant because it is implied by the constraint which holds at each call of $newsat1(X)$. Indeed, for $newsat1(X)$ in the body of clause 1 we have that $X=0$ holds, and for $newsat1(Y)$ in the body of clause 3, we have that $Y \geq 1$ holds. Thus, by applying the contextual constraint replacement rule we replace clause 2 by the fact:

5. $newsat1(X) \leftarrow$

that is, we derive a program where $newsat1(X)$ is valid. Thus, by applying rule R4s we may delete clause 3. Next, we notice that clause 4 is useless and, by applying rule R4u, we can delete it and we derive a program where $newsat2(X)$ is failed. Now, by positive and negative unfolding, from clause 1 we derive the clause:

6. $sat_{spec}(X) \leftarrow X=0.$

Since we want to derive a fact which holds for the initial state where the constraint $X=0$ is true, by applying the contextual constraint replacement rule we replace clause 6 by the fact:

7. $sat_{spec}(X) \leftarrow$

During Phase B of the specialization strategy described in Section 6, we delete redundant constraints (in our example above, the constraint $X \geq 0$ in the body of clause 2 and the constraint $X=0$ in the body of clause 6), by using the contextual constraint replacement rule R5.

During Phase C of the strategy, we derive valid and failed atoms (in our example above, $newsat1(X)$ and $newsat2(X)$, respectively). In particular, during that phase, we work bottom-up on the strata of the program (recall that at the end of Phase A we always derive a stratified program), and we simplify the definition of every predicate symbol $newp$ occurring in the program, with the aim of deriving either the fact $newp(X) \leftarrow$ or the empty definition.

6.1. The Generalization Function

Now we present the generalization function gen used during Phase A of the specialization strategy for introducing new clauses by using the constrained atomic definition rule. During the Generalization Step of Phase A, given a call pattern of the form $(c(X), sat(X,\psi))$, we introduce a new definition of the form η: $newp(X) \leftarrow gen(c(X)), sat(X,\psi)$ where $gen(c(X))$ is a constraint such that $\mathcal{D} \models \forall X\,(c(X) \rightarrow gen(c(X)))$. This condition ensures that the clause where the call pattern $(c(X), sat(X,\psi))$ occurs, can be folded by using η. Moreover, we will define $gen(c(X))$ so that it is the *least* constraint, in the sense specified below, which makes it possible to fold. This minimality condition is motivated by the fact that, as already remarked at the end of Section 2, generalization should be applied with parsimony, because it may prevent the proof of the property of interest.

An important feature of the gen function is that it has a *finite* codomain and thus, for any CTL formula ψ, a finite number of new definitions of the form $newp(X) \leftarrow gen(c(X)), sat(X,\psi)$ can be introduced. This fact ensures that the specialization strategy always terminates. As already mentioned, by doing so we obtain a method which is incomplete, in the sense that there exist properties of infinite state systems that cannot be proved. However, we will show in Section 7 that several interesting properties can indeed be proved by using the proposed generalization function.

The codomain of gen is the finite set $\mathcal{C}(E)$ of constraints constructed as follows. Let X be a variable ranging over the states of $\mathcal{K}$. We assume that every clause in $P_{\mathcal{K}} \cup \{\delta_{in}\}$ is written by using X as the first argument of the head. Thus, every clause in $P_{\mathcal{K}} \cup \{\delta_{in}\}$ is either of the form $sat(X, \psi) \leftarrow c, G$ or of the form $sat_{spec}(X) \leftarrow c, G$. We consider the set $E_{\mathcal{K}}$ of constraints e such that: (i) e is a basic constraint, (ii) there exists a clause $H \leftarrow c, G$ in $P_{\mathcal{K}} \cup \{\delta_{in}\}$ such that $solve(c, X) = e \wedge d$ for some constraint d. We assume that Property P of Section 4 holds. Let us also consider the set E_{neg} of basic constraints e' such that there exists a basic constraint $e \in E_{\mathcal{K}}$ such that the partition of $\neg e$ is of the form $c_1 \vee \ldots \vee (e' \wedge d) \vee \ldots \vee c_m$ for some constraint d. We define the following set of basic constraints: $E = E_{\mathcal{K}} \cup E_{neg}$. We identify two elements e and e' in E iff $\mathcal{D} \models \forall\, (e \leftrightarrow e')$. Let $\mathcal{C}(E)$ be the smallest set of constraints including *true*, all constraints in E, and closed w.r.t. $\wedge$. By construction, $\mathcal{C}(E)$ is a finite set.

We define $gen(c)$ as the least constraint in $\mathcal{C}(E)$ w.r.t. the implication ordering, such that $\mathcal{D} \models \forall\, (c \rightarrow gen(c))$. The constraint $gen(c)$ can be computed by applying the algorithm described below. This algorithm performs a breadth-first visit of the directed acyclic graph G which is constructed as follows: (i) the vertices of G are the constraints in E, and (ii) there exists an edge from e to e' iff (ii.1) e and e' are distinct, (ii.2) $\mathcal{D} \models \forall\, (e \rightarrow e')$, and (ii.3) there is no $d \in E$, distinct from e and e', such that $\mathcal{D} \models \forall\, ((e \rightarrow d) \wedge (d \rightarrow e'))$. Given a vertex e of G, we denote by $Reach(e)$ the set of vertices which are reachable from e.

The Algorithm for Constraint Generalization

Input: the constraint c to be generalized and the graph G.

Output: a constraint $d \in \mathcal{C}(E)$ such that (i) $\mathcal{D} \models \forall\, (c \rightarrow d)$ and (ii) for all $e \in \mathcal{C}(E)$ if $\mathcal{D} \models \forall\, (c \rightarrow e)$ then $\mathcal{D} \models \forall\, (d \rightarrow e)$.

$d := true$;
$ToBeVisited := E$;
Let $Current$ be the set of vertices of G with no incoming edges;
for each vertex $e \in Current$ **do**
  if $\mathcal{D} \models \forall\, (c \rightarrow e)$ **then**
    $d := d \wedge e$;
    $Current := Current - \{e\}$;
    $ToBeVisited := ToBeVisited - (\{e\} \cup Reach(e))$
  else
    $Current := (Current - \{e\}) \cup \{e' \in ToBeVisited \mid \text{there is an edge from } e \text{ to } e'\}$
    $ToBeVisited := ToBeVisited - \{e\}$
end-for

6.2. The Strategy for Specializing CLP Programs

Now we present the specialization strategy which we use for verifying CTL properties of reactive systems. Let $\mathcal{K}$ be a Kripke structure based on a constraint domain $\mathcal{D}$ and let $P_{\mathcal{K}}$ be the locally stratified program constructed by the Encoding Algorithm described in Section 4.

The Specialization Strategy

Input: (i) The program $P_{\mathcal{K}}$ and (ii) a constrained atom $(init(X), sat(X, \varphi))$.

Output: (i) A specialized program $P_{\mathcal{K},spec}$ and (ii) a new predicate symbol sat_{spec} such that, for all states $s \in D$, if $\mathcal{D} \models init(s)$ then $sat(s, \varphi) \in M(P_{\mathcal{K}})$ iff $sat_{spec}(s) \in M(P_{\mathcal{K},spec})$.

Phase A. We use the following three variables: (1) P_A, which denotes the output program of this phase, (2) *Defs*, which denotes the set of definitions introduced during the specialization process, and (3) *NewDefs*, which denotes the set of definitions which have been introduced but not yet unfolded. Let *Elem* be the set of elementary properties of the Kripke structure $\mathcal{K}$.

Introduce the clause $\delta_{in} : sat_{spec}(X) \leftarrow init(X), sat(X, \varphi)$ by applying the constrained atomic definition rule R1.

$P_A := \emptyset$; $Defs := \{\delta_{in}\}$; $NewDefs := \{\delta_{in}\}$;

while there exists a clause $\nu \in NewDefs$ **do**

$NewDefs := NewDefs - \{\nu\}$;

Step 1: Unfolding-Replacement.

Let Δ be the set of clauses derived by unfolding ν w.r.t. the atom in $bd(\nu)$;

while there exists a clause γ in Δ of the form $H \leftarrow c, G_1, sat(X, \psi), G_2$, where *either* ψ belongs to *Elem* *or* ψ is of the form $\neg\psi_1$ *or* ψ is of the form $\psi_1 \wedge \psi_2$ **do**

replace γ in Δ by the set of clauses derived by unfolding γ w.r.t. $sat(X, \psi)$

end-while

Let Γ be the set of clauses obtained from Δ by: (i) applying rule R4f whereby removing all clauses with an unsatisfiable constraint in the body, and (ii) applying rule R5r whereby replacing each clause of the form $H \leftarrow c, G$ by $H \leftarrow solve(c, Y), G$, where $Y = FV(c) \cap vars(\{H, G\})$;

Step 2: Generalization.

for every (possibly renamed) call pattern $(c(X), sat(X, \psi)) \in CP(\Gamma)$ **do**

if there is no clause in *Defs* whose body is $(gen(c(X)), sat(X, \psi))$

then introduce the definition η: $newp(X) \leftarrow gen(c(X)), sat(X, \psi)$ by applying rule R1;

$Defs := Defs \cup \{\eta\}$; $NewDefs := NewDefs \cup \{\eta\}$;

end-for

Step 3: Folding.

while there exists a clause γ in Γ of the form $H \leftarrow c, G_1, L, G_2$, where L is either an atom $sat(X, \psi)$ or a negated atom $\neg sat(X, \psi)$ **do**

replace γ by the clause derived by folding γ w.r.t. L using a clause in *Defs*

end-while

$P_A := P_A \cup \Gamma$

end-while

Phase B. This Phase of the specialization strategy takes the output program P_A of Phase A as input and returns a new program P_B.

$P_B := \emptyset$;

Let $\mathbf{C}$ be $\{(init(X), sat_{spec}(X))\} \cup CP(P_A)$, where $CP(P_A)$ is the set of call patterns in P_A (see Definition 3);

for every renamed apart clause γ in P_A of the form $H \leftarrow c_1, \ldots, c_n, G$, where $c_1, \ldots, c_n$ are basic constraints **do**

apply the contextual constraint replacement rule w.r.t. $\mathbf{C}$ and derive a new clause γ' by deleting, for $i = 1, \ldots, n$, the constraint c_i if, for every constrained atom $(c, Atom)$ in $\mathbf{C}$, $\mathcal{D} \models \forall\, ((c \wedge Atom = H) \rightarrow c_i)$;

$P_B := P_B \cup \{\gamma'\}$;

end-for

Phase C. This Phase of the specialization strategy takes as input the output program P_B of Phase B and returns the final, specialized program $P_{\mathcal{K},spec}$. Let $S_1, \ldots, S_n$ be a stratification of program P_B (see Lemma 6.1 below).

$P_{\mathcal{K},spec} := \emptyset$;
for $i := 1, \ldots, n$ **do**
  repeat
    $S := S_i$;
    Apply to S_i, as long as possible, the clause removal rule R4s;
    Apply to S_i, as long as possible, the positive unfolding rule R2p and the negative unfolding rule R2n w.r.t. the valid and failed atoms occurring in $S_1 \cup \ldots \cup S_i$;
    for all clauses in S_i of the form $H \leftarrow c$ **do**
      if $\mathcal{D} \models \forall(\exists Y c)$ where $Y = FV(c) - vars(H)$
      then apply the constraint replacement rule R5r and replace $H \leftarrow c$ by the fact $H \leftarrow$
    end-for
  until $S = S_i$;
  Apply the clause removal rule R4u for removing the useless clauses from S_i;
  $P_{\mathcal{K},spec} := P_{\mathcal{K},spec} \cup S_i$
end-for

The two Theorems 6.2 and 6.3 below, establish the correctness and the termination of our specialization strategy. We first need the following lemma.

Lemma 6.1. *Let P_A and P_B the output programs of Phase A and Phase B, respectively, of the specialization strategy. Then P_A and P_B are stratified.*

Proof. Program P_A is stratified w.r.t. the level mapping λ defined as follows: $\lambda(newp) = length(\psi)$, where the definition of $newp$ in $Defs$ is $newp(X) \leftarrow sat(X, \psi)$.

Indeed, by construction, for every clause γ in P_A of the form $newp(X) \leftarrow c, G$ 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 $\lambda(newq) < \lambda(newp)$.

Since in Phase B we use the contextual constraint replacement rule only, by Corollary 5.3 program P_B is stratified. □

Theorem 6.2. [Correctness of the Specialization Strategy] *Let $\mathcal{K}$ be a Kripke structure based on a constraint domain $\mathcal{D}$ and let $P_{\mathcal{K}}$ be the locally stratified program constructed by the Encoding Algorithm. Let $init(X)$ be the constraint which specifies the set of initial states and let φ be a CTL formula. By applying the specialization strategy to the input program $P_{\mathcal{K}}$ and the constrained atom $init(X), sat(X, \varphi)$, we obtain:* (i) *a specialized program $P_{\mathcal{K},spec}$ and* (ii) *a new predicate symbol sat_{spec} such that, for all states $s \in D$, if $\mathcal{D} \models init(s)$ then $sat(s, \varphi) \in M(P_{\mathcal{K}})$ iff $sat_{spec}(s) \in M(P_{\mathcal{K},spec})$.*

Proof. Let δ_{in} be the initial definition $sat_{spec}(X) \leftarrow init(X), sat(X, \varphi)$ and let s be a state such that $\mathcal{D} \models init(s)$. Let us consider the final values of $Defs$ (i.e., the set of definitions introduced during Phase A), P_A (i.e., the output program of Phase A), and P_B (i.e., the output program of Phase B). We have that:

$sat(s, \varphi) \in M(P_{\mathcal{K}})$ iff $sat_{spec}(s) \in M(P_{\mathcal{K}} \cup \{\delta_{in}\})$
  (by the definition of M, because $Def(sat_{spec}, P_{\mathcal{K}} \cup \{\delta_{in}\}) = \{\delta_{in}\}$)
iff $sat_{spec}(s) \in M(P_{\mathcal{K}} \cup Defs)$
  (because $Def(sat_{spec}, Defs) = \{\delta_{in}\}$)

iff $sat_{spec}(s) \in M(P_{\mathcal{K}} \cup P_A)$

(by Theorem 5.1)

iff $sat_{spec}(s) \in M(P_{\mathcal{K}}) \cup M(P_A)$

(because there is no predicate symbol occurring both in $P_{\mathcal{K}}$ and in P_A)

iff $sat_{spec}(s) \in M(P_A)$

(because sat is the only predicate symbol occurring in $P_{\mathcal{K}}$).

Now, we show that $sat_{spec}(s) \in M(P_A)$ iff $sat_{spec}(s) \in M(P_B)$. Let $\mathbf{C}$ be the set of constrained atoms considered at the beginning of Phase B. Since: (i) by Lemma 6.1 P_A is stratified, (ii) $\mathbf{C} \supseteq CP(P_A)$, (iii) $(init(X), sat_{spec}(X)) \in \mathbf{C}$, and (iv) $\mathcal{D} \models init(s)$, then by Corollary 5.3 we have that, $sat_{spec}(s) \in M(P_A)$ iff $sat_{spec}(s) \in M(P_B)$.

Finally, we have that $sat_{spec}(s, \varphi) \in M(P_B)$ iff $sat_{spec}(s) \in M(P_{\mathcal{K},spec})$. Indeed, during Phase C rule R1 is not applied and rule R5 is applied only in its restricted form R5r and, thus, by Theorem 5.1, we have that $M(P_B) = M(P_{\mathcal{K},spec})$. □

Theorem 6.3. *The specialization strategy always terminates.*

Proof. We prove the termination of Phase A, Phase B, and Phase C separately.

Termination of Phase A. Let us first show the termination of each application of Steps 1, 2, and 3.

Termination of Step 1. Let us consider an application of Step 1 starting from a definition ν. Let T be the tree constructed as follows: (i) the root of T is the definition ν and (ii) for any two nodes ν_1 and ν_2 in T, ν_2 is a child of ν_1 iff ν_2 is obtained by unfolding ν_1. Since the input program $P_{\mathcal{K}}$ constructed by the Encoding Algorithm is finite, each application of the unfolding rule w.r.t. an atom of the form $sat(X, \psi)$ produces a finite number of clauses. Thus, every node of T has a finite number of children. Now, we show that every path in T is finite. Let us consider the well-founded ordering $>_a$ over atoms defined as follows. For all atoms of the form $sat(X, \psi_1)$ and $sat(Y, \psi_2)$, $sat(X, \psi_1) >_a sat(Y, \psi_2)$ iff $length(\psi_1) > length(\psi_2)$. Let $>_c$ be the well-founded ordering over clauses defined as follows. For all clauses ν_1: $H_1 \leftarrow c, G_1$ and ν_2: $H_2 \leftarrow d, G_2$, $\nu_1 >_c \nu_2$ iff G_2 can be obtained from G_1 by replacing a literal L of the form A or $\neg A$ by a conjunction of literals $L_1, \ldots, L_n$ such that, for all $i = 1, \ldots, n$, L_i is of the form A_i or $\neg A_i$ and $A >_a A_i$. Now notice that, the unfolding rule is applied w.r.t. atoms of the form $sat(X, \psi)$, where either ψ belongs to $Elem$ or ψ is of the form $\neg\psi_1$ or ψ is of the form $\psi_1 \wedge \psi_2$. Moreover, by the construction of $P_{\mathcal{K}}$, this application of the unfolding rule replaces the atom $sat(X, \psi)$ by a constrained goal of the form $c, sat(X_1, \psi_1), \ldots, sat(X_k, \psi_k)$, where $k \geq 0$ and, for $i = 1, \ldots, k$, ψ_i is a proper subformula of ψ. Thus, if ν_2 is a child of ν_1 in T then $\nu_1 >_c \nu_2$. This proves that there exist no infinite paths in T and therefore the set of nodes of T is finite. Thus, also the set of clauses derived by applications of the unfolding rule during Step 1 is finite. Since we perform at most one application of the clause removal rule or the constraint replacement rule to the clauses derived by unfolding, we have that Step 1 terminates.

Termination of Steps 2 and 3. It is guaranteed by the following two facts: (i) the set Γ of clauses is finite, and (ii) every clause contains a finite number of literals in its body, and thus, there is only a finite number of call patterns.

Now we prove the termination of Phase A. The number of iterations of the outermost while-loop is equal to the number of definitions introduced during the applications of Step 2. Thus, the termination of Phase A follows from the fact that only a finite number of definitions are introduced. Indeed, every application of the constrained atomic definition rule performed at

Step 2 introduces a clause ν of the form $newp(X) \leftarrow gen(c(X)), sat(X, \psi)$ where: (i) gen is a function with a finite codomain (see Section 6.1), (ii) ψ is a subformula of the initial CTL formula φ, and (iii) the body of ν is not a variant of the body of any clause introduced by previous applications of the definition rule.

Termination of Phase B. Phase B terminates because the input program P_A is finite and in every clause there is a finite number of basic constraints.

Termination of Phase C. It follows from the following facts: (i) the input program P_B is stratified (see Lemma 6.1), and (ii) each application of a transformation rule in the repeat-loop removes either a clause or a constraint or a literal. □

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

Theorem 6.4. [Soundness of the Verification Method] *Let $\mathcal{K}$ be a Kripke structure whose initial states are specified by the constraint $init(X)$, and let φ be a CTL formula. Let $P_{\mathcal{K}}$ be the locally stratified CLP program constructed by using the Encoding Algorithm. Let the predicate sat_{spec} and the program $P_{\mathcal{K},spec}$ be the output of the specialization strategy. If the fact $sat_{spec}(X) \leftarrow$ occurs in $P_{\mathcal{K},spec}$ then $\mathcal{K}, s \models \varphi$ holds for all initial states s of $\mathcal{K}$.*

Proof. Let s be an initial state of $\mathcal{K}$, that is, $\mathcal{D} \models init(s)$. Since the fact $sat_{spec}(X) \leftarrow$ occurs in $P_{\mathcal{K},spec}$ then $sat_{spec}(s) \in M(P_{\mathcal{K},spec})$ and, by the correctness of the specialization strategy (see Theorem 6.2), we have that $sat(s, \varphi) \in M(P_{\mathcal{K}})$. Thus, by the correctness of the Encoding Algorithm (see Theorem 4.1), $\mathcal{K}, s \models \varphi$ holds. □

6.3. An Example of Application of the Specialization Strategy

Let us consider the Kripke structure $\mathcal{K}$ presented in Example 2 and let $P_{\mathcal{K}}$ be the program constructed by using the Encoding Algorithm. We will verify that $\mathcal{K}, \langle a, 0\rangle \models \neg AF\, neg$, where neg holds in a state $\langle X_1, X_2\rangle$ iff $X_2 < 0$, by proving that $sat(a, 0, \neg AF\, neg) \in M(P_{\mathcal{K}})$. We will do so by applying the specialization strategy to the input program $P_{\mathcal{K}}$ and the constrained atom $X_1 = a, X_2 = 0, sat(X_1, X_2, \neg AF\, neg)$. The specialization strategy will start off by introducing the clause:

1. $sat_{spec}(X_1, X_2) \leftarrow X_1 = a, X_2 = 0, sat(X_1, X_2, \neg AF\, neg)$

The proof of the property of interest will consist in deriving the clause:

$sat_{spec}(X_1, X_2) \leftarrow$

The generalization function gen to be used during the application of the strategy is defined as follows. The set $E_{\mathcal{K}}$ of constraints computed from $P_{\mathcal{K}} \cup \{1\}$ as indicated in Section 6.1 is $\{X_2 < 0,\ X_1 = a,\ X_2 \leq 0,\ X_2 > 0,\ X_1 = b,\ X_2 = 0\}$. The partition of $\neg X_2 < 0$ is $X_2 \geq 0$ and the partitions of the negations of the other constraints in $E_{\mathcal{K}}$ generate constraints in $E_{\mathcal{K}}$. Thus, the set E of constraints is defined as follows:

$E = E_{\mathcal{K}} \cup \{X_2 \geq 0\}$

Let $\mathcal{C}(E)$ be the closure of $\{true\} \cup E$ w.r.t. conjunction (see Section 6.1). Then, given a constraint $c(X_1, X_2)$, $gen(c(X_1, X_2))$ is the least constraint in $\mathcal{C}(E)$ w.r.t. the implication ordering, such that $\mathcal{D} \models \forall X_1 \forall X_2\, (c(X_1, X_2) \rightarrow gen(c(X_1, X_2)))$.

Let us now describe how the specialization strategy works in our example.

Phase A.
We start off by introducing the definition clause 1 in *Defs* and *NewDefs*.

First iteration.
Step 1: Unfolding-Replacement. We apply the unfolding rule to clause 1 w.r.t. the atom in its body and we derive:

2. $sat_{spec}(X_1, X_2) \leftarrow X_1 = a, X_2 = 0, \neg sat(X_1, X_2, AF\ neg)$

Step 2: Generalization. The only call pattern in clause 2 is:

$X_1 = a, X_2 = 0\,, sat(X_1, X_2, AF\ neg)$

Since, the generalization of $X_1 = a, X_2 = 0$ is $X_1 = a, X_2 = 0$ itself, we introduce the following new definition:

3. $newsat1(X_1, X_2) \leftarrow X_1 = a, X_2 = 0,\ sat(X_1, X_2, AF\ neg)$

Thus, *Defs* is {d1, d3}.

Step 3: Folding. By folding clause 2 using clause 3, we derive:

4. $sat_{spec}(X_1, X_2) \leftarrow X_1 = a, X_2 = 0, \neg newsat1(X_1, X_2)$

Now, *NewDefs* = {3} and we iterate the specialization process as follows.

Second iteration.
Step 1: Unfolding-Replacement. We apply the unfolding and constraint replacement rules to clause 3 and we derive:

5. $newsat1(X_1, X_2) \leftarrow X_1 = a, X_2 = 2,\ sat(X_1, X_2, AF\ neg)$

Step 2: Generalization. The only call pattern in clause 5 is:

$X_1 = a, X_2 = 2,\ sat(X_1, X_2, AF\ neg)$

The generalization of the constraint $X_1 = a \wedge X_2 = 2$ is $X_1 = a, X_2 > 0$ and, thus, we introduce the following new definition:

6. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_2 > 0,\ sat(X_1, X_2, AF\ neg)$

Defs is {1, 3, 6}.

Step 3: Folding. By folding clause 5 using clause 6, we derive:

7. $newsat1(X_1, X_2) \leftarrow X_1 = a, X_2 = 2, newsat2(X_1, X_2)$

Since *NewDefs* = {6} we iterate the specialization process as follows.

Third iteration.
Step 1: Unfolding-Replacement. By unfolding and constraint replacement, from clause 6 we derive:

8. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_2 > 0, X_3 = X_2 + 2, X_4 = b,$
$\qquad sat(X_1, X_3, AF\ neg),\ sat(X_4, X_2, AF\ neg)$

Step 2: Generalization. The call patterns in clause 8 are (after variable renaming):

$X_1 = a, X_2 > 2,\ sat(X_1, X_2, AF\ neg)$
$X_1 = b, X_2 > 0,\ sat(X_1, X_2, AF\ neg)$

We consider the first call pattern. The generalization of the constraint $X_1 = a, X_2 > 2$ is the constraint $X_1 = a, X_2 > 0$ and, since the constrained atom $X_1 = a, X_2 > 0,\ sat(X_1, X_2, AF\ neg)$ is the body of clause 6 in *Defs*, we do not introduce a new definition for this constrained atom. Now we consider the second call pattern in clause 8. Since the generalization of $X_1 = b, X_2 > 0$ is $X_1 = b, X_2 > 0$ itself and no clause in *Defs* has body $X_1 = b, X_2 > 0,\ sat(X_1, X_2, AF\ neg)$, we introduce the definition:

9. $newsat3(X_1, X_2) \leftarrow X_1 = b, X_2 > 0, sat(X_1, X_2, AF\ neg)$

Thus, *Defs* is {1, 3, 6, 9}.

Step 3: Folding. By folding using clauses 6 and 9, from clause 8 we derive:

10. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_2 > 0, X_3 = X_2 + 2, X_4 = b,$
$newsat2(X_1, X_3), newsat3(X_4, X_2)$

Now, *NewDefs* = {9} and we perform one more iteration of the specialization process.

Fourth iteration.
Step 1: Unfolding-Replacement. We now proceed by applying the unfolding and constraint replacement rules to clause 9 and we derive:

11. $newsat3(X_1, X_2) \leftarrow X_1 = b, X_2 > 0, X_3 = X_2 + 1,\ sat(X_1, X_3, AF\ neg)$

Step 2: Generalization. The only call pattern in clause 11 is (after variable renaming): $X_1 = b, X_2 > 1,\ sat(X_1, X_2, AF\ neg)$. The generalization of $X_1 = b, X_2 > 1$ is $X_1 = b, X_2 > 0$. Since $X_1 = b, X_2 > 0,\ sat(X_1, X_2, AF\ neg)$ is the body of clause 9 in *Defs*, we need not introduce any new definition.

Step 3: Folding. By folding clause 11 using 9, we derive:

12. $newsat3(X_1, X_2) \leftarrow X_1 = b, X_2 > 0, X_3 = X_2 + 1,\ newsat3(X_1, X_3)$

Since there are no definitions in *NewDefs* we conclude Phase A with the following program P_A:

4. $sat_{spec}(X_1, X_2) \leftarrow X_1 = a, X_2 = 0,\ \neg newsat1(X_1, X_2)$
7. $newsat1(X_1, X_2) \leftarrow X_1 = a, X_2 = 2,\ newsat2(X_1, X_2)$
10. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_2 > 0, X_3 = X_2 + 2, X_4 = b,$
$newsat2(X_1, X_3),\ newsat3(X_4, X_2)$
12. $newsat3(X_1, X_2) \leftarrow X_1 = b, X_2 > 0, X_3 = X_2 + 1,\ newsat3(X_1, X_3)$

Phase B.
We consider the set **C** consisting of the following constrained atoms:

cp1. $X_1 = a, X_2 = 0,\ sat_{spec}(X_1, X_2)$
cp2. $X_1 = a, X_2 = 0,\ newsat1(X_1, X_2)$
cp3. $X_1 = a, X_2 = 2,\ newsat2(X_1, X_2)$
cp4. $X_1 = a, X_2 > 2,\ newsat2(X_1, X_2)$
cp5. $X_1 = b, X_2 > 0,\ newsat3(X_1, X_2)$
cp6. $X_1 = b, X_2 > 1,\ newsat3(X_1, X_2)$

where the constraint in cp1 is $init(X_1, X_2)$ and {cp2, cp3, cp4, cp5, cp6} is the set $CP(P_A)$ of the call patterns in P_A (after variable renaming). We apply the contextual constraint replacement rule for deleting redundant constraints from the clauses of P_A as follows. We delete the constraint $X_1 = a, X_2 = 0$ from clause 4, because it is implied by the constraint in cp1, and we derive the following clause:

13. $sat_{spec}(X_1, X_2) \leftarrow \neg newsat1(X_1, X_2)$

We also delete the constraint $X_2 > 0$ from clause 10, because it is implied by the constraints of the call patterns cp3 and cp4 of $newsat2(X_1, X_2)$. We derive the following clause:

14. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_3 = X_2 + 2, X_4 = b,$
$newsat2(X_1, X_3),\ newsat3(X_4, X_2)$

Similarly, we remove the constraint $X_2 > 0$ from clause 12 thereby obtaining the clause:

15. $newsat3(X_1, X_2) \leftarrow X_1 = b, X_3 = X_2 + 1,\ newsat3(X_1, X_3)$

Thus, we end Phase B with program P_B consisting of clauses 13, 7, 14, and 15.

Phase C.
We compute a stratification of the program P_B and we get $P_B = S_1 \cup S_2$, where $S_1 = \{7, 14, 15\}$ and $S_2 = \{13\}$. Then, we process the two strata of P_B as follows.
Stratum S_1. Since the predicates $newsat1$, $newsat2$, and $newsat3$ are useless in S_1, we remove their definitions and we derive $S_1 = \emptyset$.
Stratum S_2. The atom $newsat1(X_1, X_2)$ is failed in the program $S_1 \cup S_2$, which contains clause 13 only. Thus, by applying the negative unfolding rule R2n to clause 13, we derive our final, specialized program $P_{\mathcal{K},spec}$ which consists of the following clause:

16. $sat_{spec}(X_1, X_2) \leftarrow$

Thus, as desired, we have proved that $\mathcal{K}, \langle a, 0\rangle \models \neg AF\, neg$ holds.

7. Examples of Protocol Verification via Specialization

Now we present the verification of three protocols by using our method based on program specialization: (i) the Bakery Protocol [23], (ii) the Ticket Protocol [2], and the Bounded Buffer Protocol [6].

The Bakery Protocol and the Ticket Protocol ensure mutual exclusion between two concurrent processes A and B trying to access a shared resource. We verified that either of these protocols: (i) indeed guarantees mutually exclusive accesses to the resource, and (ii) will eventually serve a process requesting the resource.

The Bounded Buffer Protocol governs the interaction between two message producers and two message consumers communicating through a shared buffer of limited size. We verified that no message is lost during communications.

The verification of all the temporal properties was performed automatically by using the experimental constraint logic program transformation system MAP [15].

7.1. The Bakery Protocol

The state s_A of process A is represented by a pair $\langle c_A, a\rangle$ where c_A is an element of the set $\{think, wait, use\}$ of *control states*, and a is a counter which takes as value a non negative real number (we could have used natural numbers instead, but real numbers allow us a simpler constraint solver for $\mathcal{R}_{lin}$). Analogously, the state s_B of process B is represented by a pair $\langle c_B, b\rangle$.

The evolution over time of process A is modeled by the transition relation R_A (depicted in Figure 2) which also uses the counter b associated with process B:

$$\begin{aligned} R_A \;=\; & \{(\langle think, a\rangle, \langle wait, b+1\rangle)\} \cup \\ & \{(\langle wait, a\rangle, \langle use, a\rangle) \mid a < b \text{ or } b = 0\} \cup \\ & \{(\langle use, a\rangle, \langle think, 0\rangle)\} \end{aligned}$$

The evolution over time of process B is modeled by an analogous transition relation R_B, where a and b are interchanged.
The state of the system resulting by the asynchronous parallel composition of processes A and B, is represented by the 4-tuple $\langle c_A, a, c_B, b\rangle$. Thus, the transition relation of the system is (here and in the following examples, for reasons of simplicity, we will feel free to omit some angle brackets):

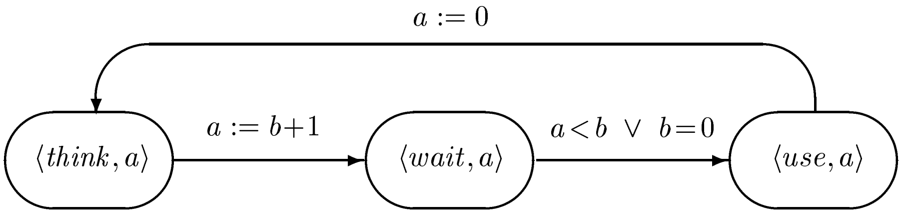

Figure 2: Process A of the Bakery Protocol.

$$R = \{(s_A, s_B, s'_A, s_B) \mid (s_A, s'_A) \in R_A\} \cup \{(s_A, s_B, s_A, s'_B) \mid (s_B, s'_B) \in R_B\}$$

This system has an infinite number of states, because counters may increase in an unbounded way, as the following computation path illustrates:

$\langle think, 0, think, 0\rangle$, $\underline{\langle wait, 1, think, 0\rangle}$, $\langle wait, 1, wait, 2\rangle$, $\langle use, 1, wait, 2\rangle$,
$\langle think, 0, wait, 2\rangle$, $\overline{\langle think, 0, use, 2\rangle}$, $\langle wait, 3, use, 2\rangle$, $\underline{\langle wait, 3, think, 0\rangle}$, ...

The set I of initial states is the singleton $\{\langle think, 0, think, 0\rangle\}$.

We have applied our specialization method to the verification of two properties of the Bakery Protocol: (i) the mutual exclusion property, and (ii) the starvation freedom property. The mutual exclusion property is a *safety* property which says that 'the system will never reach a state where both processes are using the shared resource'. The starvation freedom property is a *liveness* property which says that 'if a process wants to use a resource then it will eventually get it'. The mutual exclusion property can be expressed by the CTL formula $\neg EF\, unsafe$, where *unsafe* is an elementary property which holds iff both processes are in control state *use*, that is,

for all states $s \in S$, $unsafe \in L(s)$ iff s is of the form $\langle use, a, use, b\rangle$

where a and b are non negative real numbers.

The starvation freedom property for a process, say process A, can be expressed by the CTL formula $\neg EF(wait \wedge \neg AF\, use)$. The elementary properties *wait* and *use* hold are defined as follows:

for all states $s \in S$, $wait \in L(s)$ iff s is of the form $\langle wait, a, c_B, b\rangle$, and

for all states $s \in S$, $use \in L(s)$ iff s is of the form $\langle use, a, c_B, b\rangle$

where a and b are non negative real numbers and c_B is an element of $\{think, wait, use\}$.

7.2. The Ticket Protocol

The Ticket Protocol [2] provides an alternative solution to the mutual exclusion problem. The interaction of the two processes A and B is controlled by a process C which assigns *tickets* to A and B.

The states of the processes A and B are represented as for the Bakery Protocol. The state s_C of process C is represented by a pair $\langle t, n\rangle$ of non negative real numbers, where t is used for assigning a new ticket to A or B, and n provides an upper bound for the value of the tickets required for accessing the critical section.

The overall system is $(A\,|\,C)\,||\,(B\,|\,C)$ where $|$ denotes the synchronous parallel composition and $||$ denotes the asynchronous one. The transitions for $(A\,|\,C)$ are specified by the following relation $R_{A\,|\,C}$ (see also Figure 3):

$$\begin{aligned} R_{A\,|\,C} \;=\; & \{(\langle think, a, t, n\rangle\,, \langle wait, t, t+1, n\rangle)\} \cup \\ & \{(\langle wait, a, t, n\rangle\,, \langle use, a, t, n\rangle) \mid a \le n\} \cup \\ & \{(\langle use, a, t, n\rangle\,, \langle think, 0, t, n+1\rangle)\} \end{aligned}$$

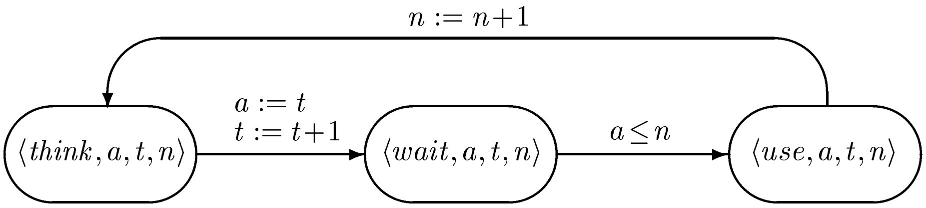

Figure 3: The Ticket Protocol: $(A \,|\, C)$.

The transitions for $(B \,|\, C)$ can be specified by a relation $R_{B\,|\,C}$, which is obtained replacing a by b in $R_{A\,|\,C}$.
The state of the overall system is represented by the 6-tuple $\langle c_A, a, c_B, b, t, n\rangle$ and its transition relation is the following:

$$\begin{aligned} R = \{&(s_A, s_B, s_C, s'_A, s_B, s'_C) \mid (s_A, s_C, s'_A, s'_C) \in R_{A\,|\,C}\} \cup \\ \{&(s_A, s_B, s_C, s_A, s'_B, s'_C) \mid (s_B, s_C, s'_B, s'_C) \in R_{B\,|\,C}\} \end{aligned}$$

This system has an infinite number of states, because there is no upper bound to the values of t and n.

The set I of the initial states is $\{\langle think, 0, think, 0, t, n\rangle \mid t{=}n\}$.

We have applied our verification method for proving the mutual exclusion property and the starvation freedom property of the Ticket Protocol. The mutual exclusion property can be expressed by the CTL formula $\neg EF\, unsafe$. The elementary property *unsafe* is defined as follows:

for all states $s \in S$, $unsafe \in L(s)$ iff s is of the form $\langle use, a, use, b, t, n\rangle$

where a, b, t and n are non negative real numbers.

The starvation freedom property for a process, say process A, can be expressed by the CTL formula $\neg EF(wait \wedge \neg AF\, use)$. The set of states where the elementary properties *wait* and *use* hold can be defined as follows:

for all states $s \in S$, $wait \in L(s)$ iff s is of the form $\langle wait, a, c_B, b, t, n\rangle$, and

for all states $s \in S$, $use \in L(s)$ iff s is of the form $\langle use, a, c_B, b, t, n\rangle$

where a, b, t, and n are non negative real numbers and c_B is an element of $\{think, wait, use\}$.

7.3. The Bounded Buffer Protocol

The Bounded Buffer Protocol governs the interaction of five processes: two *producers* P_1, P_2, two *consumers* C_1, C_2 and the buffer B.

The state sp_i of process P_i, where $i \in \{1,2\}$, is represented by a real number p_i which is the number of messages produced by P_i during the protocol run. Analogously, the state sc_i of process C_i, where $i \in \{1,2\}$, is described by a real number c_i which is the number of messages consumed by C_i during the protocol run. The state b of the buffer B is described by a pair $\langle S, A\rangle$ of real numbers where S denotes the buffer size (which does not change over time) and A denotes the number of available locations.

The overall system is $(P_1 \,|\, B) \,||\, (P_2 \,|\, B) \,||\, (C_1 \,|\, B) \,||\, (C_2 \,|\, B)$ where the transitions for $(P_i \,|\, B)$, where $i \in \{1,2\}$, are specified by the following relation:

$R_{P_i\,|\,B} = \{(\langle p_i, S, A\rangle, \langle p_i + 1, S, A-1\rangle) \mid A > 0\}$

and the transitions for $(C_i \,|\, B)$, where $i \in \{1,2\}$, are specified by the following relation:

$R_{C_i \mid B} = \{(\langle c_i, S, A\rangle, \langle c_i + 1, S, A + 1\rangle) \mid A < S\}$

The state of the overall system is represented by the 5-tuple $\langle sp_1, sp_2, sc_1, sc_2, b\rangle$ and its transition relation is the following:

$$R = \left(\bigcup_{i\in\{1,2\}} \{(sp_1, sp_2, sc_1, sc_2, b) \mid (sp_i, b, sp'_i, b') \in R_{P_i \mid B}\}\right) \cup$$
$$\left(\bigcup_{i\in\{1,2\}} \{(sp_1, sp_2, sc_1, sc_2, b) \mid (sc_i, b, sc'_i, b') \in R_{C_i \mid B}\}\right)$$

This system has an infinite number of states, because sp_1, sp_2, sc_1 and sc_2 do not have an upper bound.

The set S_0 of initial states is $\{\langle 0, 0, 0, 0, (S, A)\rangle \mid A{=}S\}$.

We have applied our verification method for proving that no message is lost during the evolution of the system, that is, 'the number of non empty locations in the buffer is equal to the number of messages produced and not consumed'.

This property can be expressed by the CTL formula $\neg EF\, lost$, where *lost* can be defined as follows:

for all states $s \in S$ of the form $\langle sp_1, sp_2, sc_1, sc_2, (S, A)\rangle$,
$lost \in L(s)$ iff $(S-A > sp_1+sp_2-sc_1-sc_2) \vee (S-A < sp_1+sp_2-sc_1-sc_2)$.

8. Related Work and Conclusions

In recent years many logic-based techniques have been developed for automatically verifying properties of reactive systems, the most successful of them being model checking [9]. The success of model checking is also due to the use of Binary Decision Diagrams which provide a very compact symbolic representation of a possibly very large, but finite, set of states. In order to overcome this finiteness restriction, some efforts have recently been devoted for dealing with infinite state systems by incorporating into model checking some abstraction and deduction techniques (see [39] for a brief survey).

Recent papers also demonstrate the usefulness of logic programming and constraint logic programming as a basis for the verification of finite or infinite state systems.

In [34] the authors present XMC, a model checking system implemented in the tabulation-based logic programming language XSB[36]. XMC can verify temporal properties expressed in the alternation-free fragment of the μ-calculus of finite state reactive systems specified in a CCS-like language. The XMC implementation contains many source-level optimizations which take advantage of the tabulation-based execution mechanism of XSB, thereby achieving performances comparable to those of state-of-the-art model checkers.

A method for the verification of some CTL properties of infinite state concurrent systems using constraint logic programming is described in [11]. Depending on the formula and the system being verified, suitable CLP programs are introduced. The truth of CTL properties is then verified by computing exact and approximated least and greatest fixed points of those programs, but unfortunately there is no guarantee of termination.

In [25] the authors show that a restricted form of partial deduction of logic programs, augmented with abstract interpretation, is sufficient to solve all coverability problems of infinite state Petri nets. Moreover, it is shown how it is possible to compute the Karp-Miller tree and Finkel's minimal coverabilty set, by using partial deduction algorithms.

In [32] a model checker is presented for verifying CTL properties of finite state systems, by using CLP programs over finite constraint domains which 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 which uses constructive negation and tabled resolution.

In [18] an automatic method for verifying safety properties of infinite state Petri nets with parametric initial markings is presented. The method tries to construct the reachability set of the Petri net being verified by computing the least fixpoint of CLP with Presburger arithmetic constraints. Invariant checking and transformations of Petri nets are used for improving performance.

A method for proving safety and liveness properties for parameterized finite state systems with various network topologies is presented in [35]. The verification process is carried out by proving goal equivalence in logic programs using unfold/fold based program transformation.

Our paper presents a systematic method for verifying CTL properties of infinite state concurrent systems based on a variant of the techniques developed in [17] for specializing constraint logic programs. The main features by which our method may show some advantages w.r.t. the above-mentioned approaches are: (i) we consider *infinite* state concurrent systems [38] whose transitions can be specified by constraints over a generic domain, (ii) we verify properties specified by using any CTL formula, and (iii) our verification method terminates in all cases.

We have applied our verification method to the familiar examples of: the Bakery Protocol [23], the Ticket Protocol [2], and the Bounded Buffer Protocol. We have proved that the first two protocols ensure mutual exclusion and starvation freedom. We have also proved that no message is lost when complying with the Bounded Buffer Protocol.

We believe that the use of CLP as modeling language together with program specialization as inference system, provides a very flexible and powerful tool for the verification of infinite state systems. Indeed, constraints allow simple representations of infinite sets of values, and the declarativeness of logic programming makes it easy to model a large variety of systems and properties.

Acknowledgements

We would like to thank Giorgio Delzanno, Sandro Etalle, and Michael Leuschel for very helpful comments on drafts of this work.

References

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

[2] G. R. Andrews, *Concurrent programming: principles and practice.* Addison-Wesley, 1991.

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

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

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

[6] T. Bultan, R. Gerber, and W. Pugh, "Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 747–789, 1999.

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

[8] 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.

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

[10] 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.

[11] G. Delzanno and A. Podelski, "Model checking in CLP," in *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)* (R. Cleaveland, ed.), Lecture Notes in Computer Science 1579, pp. 223–239, Springer-Verlag, 1999.

[12] 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*, Lecture Notes in Computer Science 85, (Berlin), pp. 169–181, Springer-Verlag, 1981.

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

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

[15] F. Fioravanti, "MAP: A system for transforming constraint logic programs." available at `http://www.iasi.rm.cnr.it/~fioravan`, 2001.

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

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

[18] L. Fribourg and H. Olsén, "Proving safety properties of infinite state systems by compilation into Presburger arithmetic," in *CONCUR '97*, Lecture Notes in Computer Science 1243, pp. 96–107, Springer-Verlag, 1997.

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

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

[21] 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.

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

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

[24] M. Leuschel, "Logic program specialisation," in *Partial Evaluation - Practice and Theory* (J. Hatcliff and P. T. E. T. Mogensen, eds.), Lecture Notes in Computer Science 1706, pp. 155–188, Springer, 1998.

[25] M. Leuschel and H. Lehmann, "Solving coverability problems of Petri nets by partial deduction.," in *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, (N.Y.), pp. 268–279, ACM Press, Sept. 20–23 2000.

[26] M. Leuschel and T. Massart, "Infinite state model checking by abstract interpretation and program specialization," in *Proceedings of LOPSTR '99, Venice, Italy* (A. Bossi, ed.), Lecture Notes in Computer Science 1817, pp. 63–82, Springer, 1999.

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

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

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

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

[31] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 1703, pp. 219–233, Springer, 1999.

[32] U. Nilsson and J. Lübcke, "Constraint logic programming for local and symbolic model-checking," in *CL 2000: Computational Logic* (J. W. Lloyd *et al.*, eds.), Lecture Notes in Artificial Intelligence 1861, pp. 384–398, 2000.

[33] 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, 1987.

[34] 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 *CAV '97*, Lecture Notes in Computer Science 1254, pp. 143–154, Springer-Verlag, 1997.

[35] 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.

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

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

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

[39] N. Shankar, "Combining theorem proving and model checking through symbolic analysis," in *CONCUR 2000: Concurrency Theory*, no. 1877 in Lecture Notes in Computer Science, (State College, PA), pp. 1–16, Springer-Verlag, Aug. 2000.

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

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