Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Informatica

XIII Ciclo – 2002– XIII-02-2

Transformation of Constraint Logic Programs for Software Specialization and Verification

Fabio Fioravanti

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Informatica

XIII Ciclo - 2002– XIII-02-2

Fabio Fioravanti

Transformation of Constraint Logic Programs for Software Specialization and Verification

Thesis Committee

Prof. Benedetto Intrigila (Advisor)
Prof. Eugenio Omodeo
Dr. Maurizio Proietti

Reviewers

Prof. Sandro Etalle
Prof. Michael Leuschel

Author's address:

Fabio Fioravanti
Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti"
Consiglio Nazionale delle Ricerche
Viale Manzoni 30, I-00185 Roma, Italy
e-mail: fioravanti@iasi.rm.cnr.it
www: http://www.iasi.rm.cnr.it/~fioravan

Abstract

In this thesis we will develop a methodology for *transforming constraint logic programs* and we will demonstrate its effectiveness by using it for (i) the *specialization* of a program to the context of use and (ii) the *verification* of temporal properties of infinite state concurrent systems.

We will introduce new transformation rules which allow us to perform program optimizations which cannot be done by using the transformation rules already presented in the literature, and highly parameterized strategies which guide the application of the transformation rules. We will show the semantic correctness of the transformation rules and the termination of the strategies.

Acknowledgements

Many people contributed, in different ways, to the writing of this thesis.

Firstly, I would like to thank Maurizio Proietti for his invaluable help and guidance through many passages along the way to this thesis. His patience and generosity allowed me to grow as a researcher, and his persistent faith in my abilities helped me to overcome my doubts. It is not excessive to say that the present thesis would not have been possible without his support.

I want to express my sincere gratitude to Alberto Pettorossi for his friendly advice on many matters over the years. Alberto has always been encouraging and he provided me with new research ideas and relevant references.

I am very grateful to my reviewers, Sandro Etalle and Michael Leuschel, for their helpful comments on a draft of this thesis.

I want to thank Michael Leuschel for his invitation to visit the Department of Electronics and Computer Science of the University of Southampton. It was a very stimulating and rewarding experience.

Thanks also to the coordinator of the doctoral course, Prof. Rossella Petreschi, and to a colleague of mine, Irene Finocchi, for their help in doing bureaucratic stuff.

During my doctoral studies I have got the opportunity to meet several persons who provided me with useful advice. The following is a non-exhaustive list: Michael Butler, Giorgio Delzanno, Stefan Gruner, Benedetto Intrigila, Paola Inverardi, Enrico Nardelli, Ulrich Ultes-Nitsche, Guido Proietti, Giovanni Rinaldi, Natarajan Shankar, Enrico Tronci, Moshe Vardi and Marisa Venturini Zilli.

I acknowledge financial support from the following institutions and organizations: IASI-CNR, the University of Rome "La Sapienza", the University of Southampton, the CP2001 Doctoral Programme and the Association for Logic Programming.

On a personal level, I am indebted to my girlfriend, Lavinia, for her support and understanding. Her love keeps me up and brings joy in my life.

Finally, I want to express my deepest gratitude to my family for continuous support over the years.

This thesis is dedicated to my family and to the memory of my grandparents.

Contents

Chapter 1

Introduction

In this thesis we will develop a methodology for *transforming constraint logic programs*. We will focus on the following two applications: (i) the *specialization* of a program to the context of use and (ii) the *verification* of temporal properties of infinite state concurrent systems.

1.1 Program Transformation

We consider the program transformation methodology based on rules and strategies as described in [13, 60, 80]. The process of deriving programs by transformation can be formalized as the construction of a sequence $P_0, \ldots, P_n$ of programs where, for $k = 0, \ldots, n-1$, program P_{k+1} is obtained from program P_k by applying a semantics preserving transformation rule. Thus, if the initial program P_0 is correct w.r.t. a given specification, then also the final program P_n is correct w.r.t. the same specification.

The transformation is effective if the final program is more efficient than the initial program. However, the undisciplined application of the transformation rules gives no warranty about performance improvements, hence transformation rules have to be applied according to suitable transformation strategies. Strategies guide the application of the transformation rules with the goal of reducing nondeterminism and avoiding redundant computations such as multiple visits of data structures. By achieving these goals, program efficiency is improved.

The advantage of the approach based on rules and strategies consists in the fact that it allows us to separate the issue of deriving a correct program from that of deriving an efficient program.The program transformation methodology has been developed in a number of different language paradigms, such as, functional programming [13], logic programming [80] and constraint logic programming [9, 25, 51].

1.2 Constraint Logic Programming

We will be concerned with the development of automatic techniques for the transformation of programs written in a constraint logic language.

Constraint Logic Programming [38] extends the usual logic programming framework [49] by allowing constraints over a generic domain $\mathcal{D}$. It defines a class CLP($\mathcal{D}$) of constraint logic programming (CLP, for short) languages which is parameterized w.r.t. the constraint domain $\mathcal{D}$, that is, the mathematical structure over which computation is performed. The advantage of such an extension is twofold. From a theoretical point of view, the use of CLP generalizes several extensions of the logic programming paradigm in a uniform framework. From a practical point of view, the CLP framework allows us to use efficient algorithms specifically developed for the constraint domain $\mathcal{D}$ under consideration (e.g., Fourier's variable elimination method for linear inequalities over the reals).

In the first part of this thesis we will apply the transformation of constraint logic programs to *program specialization*. The second part will be devoted to the verification of infinite state concurrent systems.

1.3 Program Specialization

Program specialization is a powerful methodology for software engineering and, in particular, for program reuse. Program specialization consists in a source-to-source program transformation whose goal is to adapt a generic program to the specific context where it has to be used. This adaptation process may be done via automatic or semiautomatic techniques. One such technique is *partial evaluation* [40]. As illustrated in Figure 1.3.1, program specialization takes as input a program P and part of its input data and produces as output a *residual* program Q such that running Q on the remaining part of the input data produces the same result as running P on all its input data.

Via program specialization one can perform many sophisticated program optimizations by taking advantage of the contexts where programs are used. In particular, it is possible to avoid run-time computations which depend on the known part of the input. Thus, program specialization is a very effective technique for program reuse. Indeed, it allows the programmer to write a single parameterized general program, which is usually easy to understand and to maintain, instead of writing many different programs sharing many similar computations, and each tailored to a different use. The task of generating efficient programs from the general one, which is often not so efficient, is left to the program specialization process.

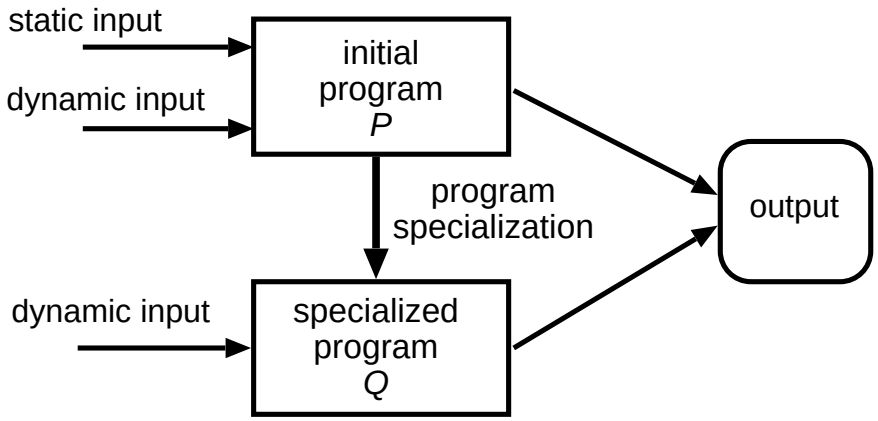

Figure 1.3.1: Program Specialization

Program specialization has been proposed and studied for various programming languages including imperative programming languages like C [3] and Fortran [8], as well as declarative programming languages like functional programming [40], logic programming [32, 43, 50, 60], constraint logic programming [84] and functional logic programming [2] languages. Successful applications of program specialization include ray tracing, Fast Fourier Transform, program compilation and compiler generation.

In this thesis we will propose a technique for specializing constraint logic programs, called *contextual specialization*. Our techniques follows the program transformation approach based on rules and strategies. We will present a set of transformation rules and fully automatic strategies for the specialization of constraint logic programs over a generic constraint domain $\mathcal{D}$. In particular, we will adapt some of the unfold/fold rules considered in [9, 25, 51] and we will introduce new transformation rules which allow us: (i) to perform program optimizations which cannot be done by using the transformation rules already presented in the literature, and (ii) to specialize constraint logic programs with locally stratified negation (see Section 3.1) w.r.t. the properties of the input data. Our automatic strategies for contextual specialization of CLP($\mathcal{D}$) programs generalize the strategies for the partial evaluation of logic programs presented in [35, 64, 72] and they use concepts borrowed from the fields of constraint programming, partial evaluation, rewrite systems, and abstract interpretation. In particular, our strategies are parameterized with respect to: (i) suitable *solvers* for simplifying constraints [38], (ii) *well-quasi orders* [21, 44] for ensuring the termination of the unfolding process and for activating the clause generalization process, and (iii) *widening operators* [17] for ensuring the termination of that generalization process.

We will show: (1) the correctness of our transformation rules w.r.t. the least $\mathcal{D}$-model in the case of definite CLP programs [39], (2) the correctness of our transformation rules w.r.t. the perfect model in the case of programs with locally stratified negation [6, 65], and (3) the termination of our program specialization strategies.

1.4 Verification of Concurrent Systems

We will also study how to apply the techniques for transforming constraint logic programs to the automatic verification of temporal properties of finite or infinite state concurrent systems.

A *concurrent system* can be informally defined as a set of components, called *processes*, which run in parallel and communicate with each other. Each process executes a sequence of statements. At any given instant of time, every process is in a state describing all its observable properties. Depending on the type of parallel composition of processes, we distinguish between *asynchronous* systems, in which exactly one process makes a step at each instant of time, and *synchronous* systems, in which all processes make a step at each instant of time. Communication can be achieved by using *message passing* or *shared variables*. When message passing is used, one process sends a message which is received by another process. When shared variables are used, one process modifies the value of a variable which can be read by another process.

In order to give a formalization of the notion of concurrent system we need to describe both its static and dynamic aspects. The static aspects are captured through the notion of *state*, which is a description of the concurrent system at a given instant of time. The dynamic aspects are captured by using the notion of *transition* which describes how the concurrent system evolves in time by specifying its state before and after a change occurs.

This motivates the choice of a formalism based on state transition systems for specifying the behaviour of concurrent systems. A *state transition system* consists of a set S, which represents the set of states of the system, equipped with a binary relation R over S, which represents the transitions that the system is allowed to make.

The goal of automated verification of concurrent systems is the design and the implementation of logical frameworks which allow one: (i) to formally specify these systems, and (ii) to prove their properties in an automatic way. These logical frameworks require formalisms both for the description of the systems and the description of their properties.

We will be interested in verifying properties of the evolution in time of concurrent systems. In order to express these properties, we will adopt a specific temporal logic, called *Computational Tree Logic* [14]. The Computational Tree

Logic (CTL, for short) does not consider a time variable explicitly, but it is powerful enough to express interesting properties such as safety properties (of the form 'the system will never reach an unwanted state') and liveness properties (of the form 'a process (which reaches a suitable state) will eventually reach a good state').

We will present a method which uses the specialization of constraint logic programs for verifying CTL properties of finite or infinite state concurrent systems. Our verification method can be applied to a large class of concurrent systems [76] and it consists of two steps. Given a concurrent system S, we construct a locally stratified constraint logic program P such that a CTL formula φ is true in a state s of S iff an atom of the form $sat(s, \varphi)$ holds in the perfect model semantics of P. Then, we check whether or not, for a given state s_0, $sat(s_0, \varphi)$ is in the perfect model of P, by specializing P w.r.t. the atom $sat(s_0, \varphi)$.

The motivation for developing an approach based on program transformation of constraint logic programs to the verification of properties of infinite state systems is twofold. From a theoretical point of view, constraints provide a very compact symbolic representation of infinite sets of states. From a more pragmatic point of view, this approach allows us to apply our verification method by using existing techniques and tools developed for transforming CLP programs.

Our verification method is incomplete but this limitation cannot be overcome because the problem of verifying properties of infinite state processes is undecidable and not semidecidable. However, we will show that a our verification method is able to verify several interesting properties of well-known concurrent systems. We will show the correctness of our verification method and we will also see (i) how it can be extended to a larger class of concurrent systems by restricting the properties which can be verified to a proper subset of CTL formulas, and (ii) how it can be applied for exploring in a backward way the state space of a concurrent system.

We will also present some results on the verification of properties of concurrent systems which arise from the parallel composition of an *arbitrary* number of *infinite* state processes. This class is strictly larger than the class of *parameterized systems* which arise from the parallel composition of an arbitrary number of *finite* state processes. Proofs of properties for this larger class have been presented, among others, in [56, 63, 77]. However, in contrast to [56, 63, 77] in our approach the parameter N representing the number of processes is *invisible*, no explicit induction on N is performed, and no abstraction of the set of processes is needed.

1.5 Overview of the Thesis

The thesis is organized as follows.

- In Chapter 2 we develop a methodology for specializing definite constraint logic programs, that is, CLP programs without negated atoms in their bodies. We start by providing a gentle introduction to the syntax and the semantics of constraint logic programs. We introduce a set of transformation rules and we show their correctness w.r.t. the least $\mathcal{D}$-model semantics. We define an automatic, parameterized strategy for specializing definite CLP programs and we show its correctness and its termination. The strategy is illustrated through an extended example of program specialization. The chapter ends with the presentation of some experimental results and a comparison of our methodology with related approaches presented in the literature.

- In Chapter 3 we extend the methodology presented in Chapter 2 for specializing general constraint logic programs with locally stratified negation. We extend the syntax of constraint logic programs and we consider the perfect model semantics. We introduce new transformation rules and an automatic strategy which are tailored to general CLP programs. We show that the rules and the strategy preserve the perfect model semantics and that the strategy terminates. At the end of the chapter we compare our work to related work on transformation of general (constraint) logic programs.

- In Chapter 4 we define a method for verifying CTL properties of concurrent systems which uses CLP program transformation. We start by presenting the syntax and the semantics of Computational Tree Logic. We define a class of concurrent systems and we illustrate our Encoding Algorithm for constructing a locally stratified constraint logic program whose perfect model specifies the truth of CTL formulas in a concurrent system. Then, we show how to check whether or not a concurrent system satisfies a CTL formula by applying a transformation strategy tailored to verification. Our method is illustrated by applying it to the verification of mutual exclusion and starvation freedom properties of the bakery protocol [42], the ticket protocol [4] and the bounded buffer protocol [12]. We show how to extend our verification method to a larger class of concurrent systems and how it can be applied for performing backwards verification of safety properties. The chapter ends with a comparison of our verification method with related approaches to verification based on (constraint) logic programming.

- In Chapter 5 we improve on the method presented in Chapter 4 and we show how to verify properties of concurrent systems with an arbitrary number of infinite state processes. First we introduce constraint logic programs where the constraint theory is the weak monadic second order logic of k successors, called CLP(WSkS) programs. Then we show how to use CLP(WSkS) programs to express safety properties of concurrent systems. By transforming CLP(WSkS) programs we prove the mutual exclusion property for the N-process bakery protocol. We conclude the chapter by comparing our approach with related work on the verification of concurrent systems with an arbitrary number of processes.

- In Appendix A we describe some issues related to the design, implementation and use of the MAP transformation system, which has been used to experimentally evaluate the proposed methodologies. Appendix B contains the source code for some programs mentioned in this thesis.

Chapter 2

Contextual Specialization of Constraint Logic Programs

In this chapter we address the problem of automating some techniques for the *contextual specialization* of constraint logic programs over a generic constraint domain $\mathcal{D}$ [38].

Contextual specialization is defined as follows. Given a CLP($\mathcal{D}$) program P and a constrained atom c, A derive a program P_s and an atom A_s such that, for every valuation ν we have that:

(*Contextual Specialization*)

$$\text{if } \mathcal{D} \models \nu(c) \text{ then } \quad \nu(A) \in lm(P, \mathcal{D}) \text{ iff } \nu(A_s) \in lm(P_s, \mathcal{D})$$

where $lm(P, \mathcal{D})$ denotes the *least $\mathcal{D}$-model* of P [39].

Contextual specialization is more general than the partial evaluation of CLP($\mathcal{D}$) programs based on Lloyd and Shepherdson's approach [48, 50, 84]. Indeed, partial evaluation is defined as follows. Given a CLP($\mathcal{D}$) program P and a constrained atom c, A derive a program P_{pe} and an atom A_{pe} such that, for every valuation ν we have that:

(*Partial Evaluation*)

$$\mathcal{D} \models \nu(c) \text{ and } \nu(A) \in lm(P, \mathcal{D}) \quad \text{iff} \quad \nu(A_{pe}) \in lm(P_{pe}, \mathcal{D})$$

Now we present a very simple example which illustrates the difference between contextual specialization of CLP programs and partial evaluation. More significant examples and experimental results will be discussed in Sections 2.8 and 2.9. Let us consider the following CLP($\mathcal{R}$) program P over the domain $\mathcal{R}$ of real numbers:

$p(X) \leftarrow X \geq 0,\ q(X)$ (Program P)

where q is a predicate which does not depend on p. By contextual specialization of P w.r.t. the constrained atom $X \geq 3, p(X)$ we derive the program P_s:

$p_s(X) \leftarrow q(X)$ (Program P_s)

together with the atom $p_s(X)$.

Instead, by partial evaluation of program P w.r.t. the same constrained atom $X \geq 3, p(X)$ we derive the program P_{pe}:

$p_{pe}(X) \leftarrow X \geq 3,\ q(X)$ (Program P_{pe})

together with the atom $p_{pe}(X)$.

Thus, the partially evaluated program P_{pe} is less efficient than the program P_s derived by contextual specialization, because P_{pe} redundantly checks whether or not the constraint $X \geq 3$ holds.

We perform program specialization by applying a program transformation method based on the *rules + strategies* approach [13, 60, 80].

The process of *specializing* a given program P whereby deriving program P_s, can be formalized as the construction of a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where $P_0 = P$, $P_n = P_s$ and, for $k = 0, \ldots, n-1$, program P_{k+1} is obtained from program P_k by applying one of the following transformation rules: *constrained atomic definition*, *unfolding*, *constrained atomic folding*, *clause removal*, and *contextual constraint replacement*. We will also apply the *constraint replacement rule* (see rule R5r) which is an instance of the contextual constraint replacement rule (see rule R5). These rules are illustrated below in Section 2.2.

The transformation sequence is automatically generated by applying (an instance of) a highly parameterized strategy which generalizes the strategies presented in [35, 64, 72] for the partial evaluation of definite logic programs.

In Section 2.4 we address various issues concerning the full automation of our strategies, which are described in Section 2.5. In particular, we consider the problems of: (i) when and how to unfold, (ii) when and how to generalize, and (iii) when and how to apply the contextual constraint replacement rule. Our automatic strategy for contextual specialization of CLP($\mathcal{D}$) programs is based on concepts borrowed from the fields of constraint programming, partial evaluation, and abstract interpretation [17]. In particular, we consider: (i) suitable *solvers* for simplifying constraints [38], (ii) *well-quasi orders* for ensuring the termination of the unfolding process and for activating the clause generalization process [44, 46, 79], and (iii) *widening operators* [17] for ensuring the termination of that generalization process.

We now introduce some preliminaries on the syntax and the semantics of constraint logic programs.

2.1 Constraint Logic Programming

The class CLP($\mathcal{D}$) of constraint logic programming languages is a generalization of the logic programming paradigm, and of several of its extensions,

which combines the declarativeness of logic programming with the efficiency of domain specific algorithms.

Let us begin by presenting some preliminary notions on constraint logic programming and notational conventions which will hopefully be used consistently in the rest of this thesis. For notions not defined here the reader may refer to [5, 38, 49]. An elementary presentation of first order logic is given in [57].

2.1.1 Syntax of Constraint Logic Programs

We consider a first order language $\mathcal{L}$ which is generated by an alphabet consisting of:

- an infinite set *Vars* of *variables*,
- a set Φ of *function symbols*,
- a set Π_c of *constraint predicate symbols*,
- an infinite set Π_u of *user defined predicate symbols*,

where *Vars*,Φ, Π_c and Π_u are pairwise disjoint sets.

Every function and predicate symbol has an associated *arity*, a natural number indicating how many arguments it takes. A symbol with associated arity 0 is called a *nullary* symbol. A nullary function symbol is called a *constant*. A nullary predicate symbol is called a *proposition*. A symbol with associated arity n is said to be *n-ary*.

Variables are denoted by upper case Latin letters $X, Y, \ldots$, possibly with subscripts. When no confusion arises, we will feel free to use upper case Latin letters $X, Y, \ldots$ to denote sets or sequences of variables.

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 function symbol in Φ and $t_1, \ldots, t_n$ are terms. Terms are denoted by lower case Latin letters $t, u, \ldots$. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a symbol in $\Pi_c \cup \Pi_u$ and and $t_1, \ldots, t_n$ are terms. A *formula* of $\mathcal{L}$ is either an atomic formula or a formula constructed, as usual, from formulas by means of connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$) and quantifiers ($\exists$, $\forall$).

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 $X_1, \ldots, X_n$.

Similarly, we will write the formula $p(t_1, \ldots, t_m)$ also as $p(t)$, where t denotes the sequence $t_1, \ldots, t_m$ of terms.

We assume that the set Π_c of constraint predicate symbols contains the equality symbol '='. Elements of Π_c are denoted by lower case Latin letters $c, d, e, \ldots$. A *primitive constraint* is an atomic formula of the form $c(t_1, \ldots, t_m)$ where c is a predicate symbol in Π_c and $t_1, \ldots, t_m$ are terms. 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. A constraint is denoted by $c(t_1, \ldots, t_m)$, or $c(X_1, \ldots, X_m)$ or $c(X)$. When we do not want to specify the variables occurring in a constraint we write it as c.

User defined predicate symbols are denoted by lower case Latin letters $p, q, \ldots$. An *atom* is an atomic formula of the form $p(t_1, \ldots, t_m)$ where p is a predicate symbol in Π_u and $t_1, \ldots, t_m$ are terms. Atoms are denoted by upper case Latin letters $H, A, B, \ldots$, possibly with subscripts. A *goal* is a (possibly empty) conjunction of atoms. Goals are denoted by G, possibly with subscripts.

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.

A *constrained atom* is the conjunction of a constraint and an atom. A *constrained goal* is the conjunction of a constraint and a goal, and it is denoted by K, possibly with subscripts. The empty conjunction of constraints or atoms is identified with *true*.

A *definite clause*, or *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 $H \leftarrow \mathit{true}$ are called *facts*, and they are also written as $H \leftarrow$. A clause is *constraint-free* iff no constraints occur in its body. Clauses are denoted by lower case Greek letters $\gamma, \delta, \ldots$

A *definite constraint logic program*, or *constraint logic program*, or *program*, is a finite set of clauses. Programs are denoted by the letters $P, Q, \ldots$, possibly

with subscripts.

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:

$p \leftarrow q, r$
$q \leftarrow p$
$r \leftarrow$

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 syntactic expression φ returns the syntactic expression $\rho(\varphi)$, called a *variant* of φ, obtained by replacing each 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.

Given the clause γ of the form: $H \leftarrow K_1, K_2$, where K_1 and K_2 are constrained goals, the set of the *linking variables* of K_1 in γ is the set $FV(K_1) \cap FV(H, K_2)$. Similarly, we define the set of the linking variables of a constraint or a constraint atom in a clause.

We will feel free to apply to clauses the following transformations which, as the reader may verify, preserve program semantics (see Sections 2.1.2 and 3.1):
(1) application of variable renaming,
(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.

2.1.2 Semantics of Constraint Logic Programs

The semantics of constraint logic programs is based on the notion of constraint domain.

For the given set Φ of function symbols and set Π_c of predicate symbols, a *constraint domain* $\mathcal{D}$, consists of two elements:

- a non-empty set D, which is called *carrier*, and

- a *pre-interpretation* which assigns (i) a *relation* over D^n (that is, a subset of D^n) to each n-ary constraint predicate symbol in Π_c, and (ii) a function f_D from D^n to D to each n-ary function symbol f in Φ.

In particular, the pre-interpretation 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 Π_c, we consider the satisfaction relation $\mathcal{D} \models \varphi$ which is defined as usually done in the first order predicate calculus.

A *valuation* is a function ν: $Vars \to D$. We extend the domain of ν to terms, constraints and atoms. Given a term t, we inductively define the term $\nu(t)$ as follows: (i) if t is a variable X then $\nu(t) = \nu(X)$, and (ii) if t is $f(t_1, \ldots, t_n)$ then $\nu(t) = f_{\mathcal{D}}(\nu(t_1), \ldots, \nu(t_n))$. Given a constraint c, $\nu(c)$ is the constraint obtained by replacing each free variable $X \in FV(c)$ by the ground term $\nu(X)$. Notice that $\nu(c)$ is a closed formula. Given an atom A of the form $p(t_1, \ldots, t_n)$, then $\nu(A)$ is the ground atom $p(\nu(t_1), \ldots, \nu(t_n))$.

A formula F_ν is a *ground instance* of a formula F if there exists a valuation ν such that $F_\nu = \nu(F)$ and $FV(F_\nu) = \emptyset$.

We define $ground(P)$ as the following set of ground clauses:

$$ground(P) = \{\nu(H) \leftarrow \nu(A_1), \ldots, \nu(A_m) \mid \nu \text{ is a valuation, } (H \leftarrow c, A_1, \ldots, A_m) \in P, \text{ and } \mathcal{D} \models \nu(c)\}$$

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 Π_u, that is, I is a subset of the set $\mathcal{B}_{\mathcal{D}}$ defined as follows:

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

Given a $\mathcal{D}$-interpretation I and a constraint-free, ground clause γ: $H \leftarrow A_1, \ldots, A_m$, 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 $A_i \notin I$.

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 $ground(P)$.

If we consider the following partial order $\subseteq$ between $\mathcal{D}$-models, we have the following result (see, for instance, Corollary 4.1 of [39]).

Theorem 1. *Every CLP($\mathcal{D}$) program has a least $\mathcal{D}$-model.*

This is a generalization of an analogous result which holds for the *least Herbrand model* of logic programs. The least $\mathcal{D}$-model of a CLP($\mathcal{D}$) program P is denoted by $lm(\mathcal{P}, \mathcal{D})$.

Let P be a CLP($\mathcal{D}$) program and $I \subseteq \mathcal{B}_{\mathcal{D}}$. The *immediate consequence operator* T_P is defined as follows.

$$T_P(I) = \{p(d) \in \mathcal{B}_{\mathcal{D}} \mid \text{for some ground instance } p(d) \leftarrow c, A_1, \ldots, A_n \text{ of a clause in } P \text{ we have } \mathcal{D} \models c \text{ and } A_i \in I \text{ for all } i = 1, \ldots n\}$$

T_P is monotonic and continuous w.r.t. set inclusion, and thus there exists the least fixpoint of T_P, denoted $lfp(T_P)$. We have the following fixpoint characterization of the least $\mathcal{D}$-model of a CLP($\mathcal{D}$) program P (see, for instance, [39]).

Theorem 2. *Let P be a CLP($\mathcal{D}$) program. Then, $lm(\mathcal{P}, \mathcal{D}) = lfp(T_P) = T_P \uparrow_\omega$.*

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

We also require that *solve* is *complete* w.r.t. satisfiability in the sense that, for any constraint c such that $Y = FV(c)$:

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

(ii) $solve(c, \emptyset) = false$ if c is *unsatisfiable*, that is, $\mathcal{D} \not\models \exists Y\, c$.

In (i) and (ii) 'if' can be replaced by 'iff' because *solve* is sound w.r.t. constraint equivalence. The soundness and the totality of the *solve* function are necessary to guarantee the correctness and the termination, respectively, of the program transformation strategies presented in this thesis, (see, for example, Sections 2.5 and 3.4). The assumption that the *solve* function is complete w.r.t. satisfiability guarantees that constraint satisfiability tests, which are required by our technique, are decidable and they can indeed 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.

2.2 Rules for Transforming Constraint Logic Programs

In this section we describe the transformation rules which we use for specializing CLP($\mathcal{D}$) programs. Some of the following rules are slight modifications of the unfold/fold rules considered in [9, 25, 51] and they are designed for performing contextual specialization.

R1. Constrained Atomic Definition. By *constrained atomic definition* (or *definition*, for short), we introduce the new clause

$$\delta: \quad newp(X) \leftarrow c, A$$

which is said to be a *definition*, where: (i) *newp* is a predicate symbol not occurring in $P_0, \ldots, P_k$, (ii) X is a sequence of distinct variables occurring in the constrained atom c, A, and (iii) the predicate symbol of A occurs in P_0. From program P_k we derive the new program P_{k+1} which is $P_k \cup \{\delta\}$.
For $i \geq 0$, $Defs_i$ is the set of definitions introduced during the transformation sequence $P_0, \ldots, P_i$. In particular, $Defs_0 = \emptyset$.

R2. Unfolding. Let $\gamma: \ H \leftarrow c, G_L, A, G_R$ be a renamed apart clause of P_k and let $\{A_j \leftarrow c_j, G_j \mid j = 1, \ldots, m\}$ be the set of *all* clauses in P_k such that the atoms A and A_j have the same predicate symbol. For $j = 1, \ldots, m$, let us consider the clause

$$\gamma_j: \quad H \leftarrow c,\ A{=}A_j,\ c_j,\ G_L, G_j, G_R$$

where $A{=}A_j$ stands for the conjunction of the equalities between the corresponding arguments. Then, by *unfolding* clause γ w.r.t. atom A, from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_j \mid j = 1, \ldots, m\}$.

R3. Constrained Atomic Folding. Let $\gamma: \ A \leftarrow c, G_L, B, G_R$ be a clause of P_k. Let $\delta: \ newp(X) \leftarrow d, B$ be a variant of a clause in $Defs_k$. Suppose that: (i) $\mathcal{D} \models \forall Y\ (c \rightarrow d)$, where $Y = FV(c, d)$, and (ii) no variable in $FV(\delta) - X$ occurs in $FV(A, c, G_L, G_R)$. By *folding* clause γ w.r.t. atom B using δ, we derive the new clause

$$\gamma_f: \quad A \leftarrow c, G_L, newp(X), G_R$$

and from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_f\}$.

In this rule R3 condition (i) may be replaced by the following weaker, but more complex condition: (i*) $\mathcal{D} \models \forall Y\ (c \rightarrow \exists Z\, d)$, where $Z = FV(d) - (X \cup vars(B))$ and $Y = FV(c, d) - Z$. However, by a suitable application of the constraint replacement rule R5r below, from clause δ we can derive a clause η of the form: $newp(X) \leftarrow (\exists Z\, d), B$ such that condition (i*) holds for γ and δ iff condition (i) holds for γ and η.

R4f. Clause Removal: Unsatisfiable Body. Let $\gamma: \ A \leftarrow c, G$ be a clause of P_k. If the constraint c is *unsatisfiable*, that is, $solve(c, \emptyset) = false$, then from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R4s. Clause Removal: Subsumed Clause. Let $\gamma: \ p(X) \leftarrow c, G$ with $(c, G) \neq true$, be a clause of P_k. If P_k contains a fact λ of the form $p(Y) \leftarrow$ then from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R4u. Clause Removal: Useless Clauses. Let Γ be the set of useless clauses in P_k. Then, by *removing useless clauses* from program P_k we derive the new program P_{k+1} which is $P_k - \Gamma$.

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

$$\mathcal{D} \models \forall X \, ((c, U{=}V) \ \rightarrow \ (\exists Y \, c_1 \leftrightarrow \exists Z \, c_2))$$

where: (i) $Y = FV(c_1) - vars(U, G)$, (ii) $Z = FV(c_2) - vars(U, G)$, and (iii) $X = FV(c, U = V, c_1, c_2) - (Y \cup Z)$. Then, we derive program P_{k+1} from program P_k by replacing clause γ by the clause: $p(U) \leftarrow c_2, G$. In this case we say that P_{k+1} has been derived from P_k by contextual constraint replacement w.r.t. $\mathbf{C}$.

The following rule is an instance of rule R5 for $\mathbf{C} = \{true, p(U)\}$.

R5r. Constraint Replacement. Let $\gamma: \ A \leftarrow c_1, G$ be a renamed apart clause of P_k. Assume that $\mathcal{D} \models \forall X (\exists Y \, c_1 \leftrightarrow \exists Z \, c_2)$ where: (i) $Y = FV(c_1) - vars(A, G)$, (ii) $Z = FV(c_2) - vars(A, G)$, and (iii) $X = FV(c_1, c_2) - (Y \cup Z)$. Then from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{A \leftarrow c_2, G\}$.

In the *contextual specialization strategy* of Section 2.5, we will make use of the above rule R5r for replacing a clause γ of the form $A \leftarrow c_1, G$ by the clause $A \leftarrow solve(c_1, X), G$, where X is the set of the linking variables of c_1 in γ.

2.3 Correctness of the Transformation Rules

In this section we enunciate the correctness w.r.t. the least $\mathcal{D}$-model of the transformation rules for definite constraint logic programs presented in Section 2.2.

Theorem 2.3.1. [Correctness of the Transformation Rules] *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, and
(ii) the contextual constraint replacement rule R5 is only applied in its restricted form R5r.
Then,

$lm(P_0 \cup Defs_n, \mathcal{D}) = lm(P_n, \mathcal{D})$

where $Defs_n$ *denotes the set of definitions introduced during the construction of* $P_0, \ldots, P_n$.

Proof. It follows from the correctness of the transformation rules for general constraint logic programs (see Theorem 3.3.10 in Section 3.3). □

In order to enunciate the theorem on the correctness of the contextual constraint replacement rule R5 w.r.t. the least $\mathcal{D}$-model we introduce the following notion of *call patterns* of a clause or set of clauses.

Definition 2.3.2. [*Call Patterns*] Given a clause γ of the form $p(X) \leftarrow d, A_1, \ldots, A_k$, with $k > 0$, the set of *call patterns* of γ, which is denoted by $CP(\gamma)$, is the set of triples $\langle solve(d, Y), A, Y \rangle$ such that: (i) A is A_j for some $j = 1, \ldots, k$, and (ii) Y denotes the linking variables of A_j in γ. The triple $\langle solve(d, Y), A, Y \rangle$ is said to be the call pattern of γ for A.

The set of call patterns of a set Γ of clauses, denoted by $CP(\Gamma)$, is the union of the sets of call patterns of the clauses in Γ, that is, $CP(\Gamma) = \bigcup_{\gamma \in \Gamma} CP(\gamma)$.

Call patterns will be used in our contextual specialization strategy below (see Section 2.5) for introducing new definitions and for applying the contextual constraint replacement rule R5.

Theorem 2.3.3. [Correctness of the Contextual Constraint Replacement Rule] *Let* $P_0, \ldots, P_n$ *be a transformation sequence such that, for all* $i = 0, \ldots, n-1$, *program* P_{i+1} *is derived from* P_i *by applying the contextual constraint replacement rule R5 w.r.t. a given set* $\mathbf{C}$ *of constrained atoms such that* $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X \rangle \in CP(P_0)\}$.
Then, for all constrained atoms $c, A \in \mathbf{C}$ *and for every valuation* ν *we have that:*

if $\mathcal{D} \models \nu(c)$ *then* $\nu(A) \in lm(P_0, \mathcal{D})$ *iff* $\nu(A) \in lm(P_n, \mathcal{D})$

Proof. It follows from the correctness of the contextual constraint replacement rule for general constraint logic programs (see Theorem 3.3.14 in Section 3.3). □

2.4 Well-Quasi Orders and Clause Generalization

In this section we introduce the notions of: (i) *well-quasi orders* over constrained goals, and (ii) *clause generalization*, which will be useful for ensuring

the termination of our program specialization strategy of Section 2.5. These notions are extensions to the case of CLP($\mathcal{D}$) programs of similar notions considered in the case of partial evaluation of functional and logic programs (see, for instance, [44, 46, 79]).

Let $\mathbf{N}$ denote the set of natural numbers.

Definition 2.4.1. [*Well-quasi order*] A *well-quasi order* (*wqo*, for short) over the set of constrained goals is a reflexive, transitive, binary relation $\preceq$ such that for every infinite sequence $\{K_i \mid i \in \mathbf{N}\}$ of constrained goals there exist two natural numbers i and j such that $i < j$ and $K_i \preceq K_j$ [21]. Given two constrained goals K_1 and K_2, if $K_1 \preceq K_2$ we say that K_1 is *embedded in* K_2.

Various examples of wqo's that are used for ensuring the termination of the unfolding process during the partial evaluation of logic and functional programs, can be found in [44, 79]. For our specialization example of Section 2.8 we will use the simple wqo $\preceq_L$ defined as follows.

Example 2.4.2. Given two constrained goals K_1 and K_2, we have that $K_1 \preceq_L K_2$ iff the leftmost atom (in the textual order) in K_1 and the leftmost atom (in the textual order) in K_2 have the same predicate symbol.

Definition 2.4.3. [*Constraint Lattice and Widening*] Given the set $\mathcal{C}$ of constraints over $\mathcal{D}$, we consider the partial order $\langle \mathcal{C}, \sqsubseteq \rangle$ such that for any two constraints c_1 and c_2 in $\mathcal{C}$, $c_1 \sqsubseteq c_2$ iff $\mathcal{D} \models \forall X(c_1 \rightarrow c_2)$ where $X = FV(c_1, c_2)$. We assume that $\langle \mathcal{C}, \sqsubseteq \rangle$ is a lattice, where: (i) the least element is *false*, (ii) the greatest element is *true*, (iii) the *least upper bound* of two constraints c_1 and c_2 is denoted by $c_1 \sqcup c_2$, and (iv) the *greatest lower bound* of two constraints c_1 and c_2 is their conjunction c_1, c_2.

A *widening operator* (see also [17]) is a binary operator ∇ between constraints such that:
(W1) $(c_1 \sqcup c_2) \sqsubseteq (c_1 \nabla c_2)$, and
(W2) for every infinite sequence $\{c_i \mid i \in \mathbf{N}\}$ of constraints, the infinite sequence $\{d_i \mid i \in \mathbf{N}\}$ of constraints where $d_0 = c_0$ and, for any $i \in \mathbf{N}$, $d_{i+1} = d_i \nabla c_{i+1}$, *stabilizes*, that is, $\exists h \in \mathbf{N}\ \forall k \geq h\ \mathcal{D} \models \forall X(d_h \leftrightarrow d_k)$ where $X = FV(d_h, d_k)$. Notice that, in general, ∇ is not commutative.

We now introduce the notion of *clause generalization*, which is based upon the widening operator ∇. It will be used in our strategy for contextual specialization to be presented in Section 2.5, for deriving from two given atomic definitions a new, generalized atomic definition.

Definition 2.4.4. [*Clause Generalization*] Given a clause α of the form $new1(U) \leftarrow c_1,\ q(X)$, and a clause η of the form $new2(V) \leftarrow c_2,\ q(X)$, where each variable in $(U, FV(c_1), V, FV(c_2))$ is in X, we define the *generalization of*

α *w.r.t.* η, denoted by $gen(\alpha, \eta)$, to be the clause: $genp(W) \leftarrow c_1 \nabla c_2,\ q(X)$, where *genp* is a new predicate symbol and W is the sequence of the distinct variables occurring in (U, V).

Example 2.4.5. [$\mathcal{R}_{lin}$: *Linear equations and inequations over the reals*] Let us consider the constraint domain $\mathcal{R}_{lin}$ of conjunctions of linear equations ($=$) and inequations ($<, \leq$) over real numbers. In this domain we may replace any existentially quantified constraint by an equivalent constraint without occurrences of $\exists$. Thus, without loss of generality, we may assume that every constraint is a conjunction of primitive constraints of the form $t_1\ op\ t_2$, where $op \in \{=, <, \leq\}$. However, in the examples below we may occasionally write $t_1 \leq t_2$ as $t_2 \geq t_1$, and $t_1 < t_2$ as $t_2 > t_1$.

$\mathcal{R}_{lin}$ is a lattice whose least upper bound operation is defined by the convex hull construction. Let us now introduce a widening operator for $\mathcal{R}_{lin}$ which we will use in our program specialization example of Section 2.8.

Given a constraint c, let $ineq(c)$ be the constraint obtained by replacing every equation $t_1 = t_2$ in c by the conjunction of the two inequations $t_1 \leq t_2, t_2 \leq t_1$. Assume that $ineq(c)$ is the conjunction of primitive constraints $c_1, \ldots, c_n$. For any constraint d, we define the widening $c \nabla d$ to be the conjunction of all c_i's, with $0 \leq i \leq n$, such that $d \sqsubseteq c_i$.

This widening operator satisfies Condition W1 because, by construction, $c \sqsubseteq c \nabla d$ and $d \sqsubseteq c \nabla d$, that is, $c \nabla d$ is an upper bound of c, d. Thus, we have that: $(c \sqcup d) \sqsubseteq (c \nabla d)$. Also Condition W2 holds for ∇, because for every constraint c and d the number of primitive constraints in $ineq(c)$ is not smaller than the number of primitive constraints in $ineq(c \nabla d)$.
Here is an example of clause generalization. Given the following two clauses:

α. $mmod_s(I, J, M) \leftarrow I = 0,\ J \geq 0,\ mmod(I, J, M)$

η. $newp(I, J, M) \leftarrow I = 1,\ J > 0,\ mmod(I, J, M)$

$gen(\alpha, \eta)$ is $genp(I, J, M) \leftarrow I \geq 0,\ J \geq 0,\ mmod(I, J, M)$, because we have that: (i) $ineq(I = 0,\ J \geq 0) = (I \geq 0,\ I \leq 0,\ J \geq 0)$ and (ii) $(ineq(I = 0,\ J \geq 0))$ ∇ $(I = 1,\ J > 0) = (I \geq 0,\ J \geq 0)$ because it is not the case that $(I = 1,\ J > 0) \sqsubseteq (I \leq 0)$.

2.5 An Automated Strategy for Contextual Specialization

We now describe the contextual specialization strategy for deriving efficient CLP($\mathcal{D}$) programs by specialization. This strategy is a generalization of the strategies for the partial evaluation of definite logic programs presented in [35, 64, 72].

Our strategy is parameterized by: (i) the function *solve* which is used for the application of the constraint replacement rule, (ii) an *unfolding function*

Unfold for guiding the unfolding process, (iii) a well-quasi order $\preceq_u$ over constrained goals which tells us when to terminate the unfolding process, (iv) a clause generalization function *gen*, with its associated widening operator ∇, and (v) a well-quasi order $\preceq_g$ over constrained atoms which tells us when to activate the clause generalization process. Once the choice of these parameters has been made, our strategy can be applied in a fully automatic way.

The contextual specialization strategy is divided into three phases.

Phase A. Phase A consists of the iteration of two procedures, called *Unfold-Replace* and *Define-Fold*, respectively. During the *Unfold-Replace* procedure we unfold the program to be specialized so to expose some initial portions of its computation, and we simplify the derived clauses by replacing inefficient constraints by some more efficient ones using the given function *solve*. The termination of this procedure is ensured by the use of the well-quasi order $\preceq_u$. We then apply the *Define-Fold* procedure and we fold the simplified clauses by using already available definitions and, possibly, some new definitions. Phase A is terminated when no new definitions need to be introduced for performing the folding steps. The termination of Phase A is ensured by the properties of the generalization function and well-quasi order $\preceq_g$ which guarantee that the set of generated definitions is finite.

Phase B. During Phase B, we apply the contextual constraint replacement rule and from each clause defining a predicate, say p, we remove the constraints which are known to hold when the clause is used. This information can be obtained by computing the least upper bound of the set of constraints which occur in the clauses containing a call of p.

Phase C. The goal of Phase C is to simplify the current program as follows: (i) by unfolding a clause w.r.t. valid and failed atoms, we remove atoms from its body or we remove the clause itself, (ii) we remove useless and subsumed clauses, and (iii) by applying the constraint replacement rule, we replace a constrained fact of the form $H \leftarrow c$ by the fact $H \leftarrow$ thereby inferring the validity of some atoms in the program.

For the formal description of our contextual specialization strategy we need to introduce the following data structures. We introduce a tree *Defstree*, called *definition tree,* whose nodes are the clauses introduced by the definition rule during program specialization. Moreover, for each clause δ in *Defstree* we introduce a tree *Utree*(δ), called *unfolding tree*. The root of *Utree*(δ) is δ itself, and the nodes of *Utree*(δ) are the clauses derived from δ by applying the unfolding and constraint replacement rules. The usual relation of *ancestor* between nodes in a tree gives us the relation of ancestor between clauses in *Defstree* and also between clauses in *Utree*(δ).

2.5.1 The Unfold-Replace Procedure

Let us first introduce some notions which will be useful in the definition of the *Unfold-Replace* procedure. Let P be a CLP program and Γ be a set of clauses.

(a) A clause of the form $H \leftarrow c, A_1, \ldots, A_n$ has a *non-failing* body iff c is satisfiable, and for $i = 1, \ldots, n$, A_i is not failed;

(b) an *unfolding function* is a partial function *Unfold*, which takes a clause λ and an unfolding tree T of which λ is a leaf, and returns the set $Unfold(\lambda, T)$ of clauses obtained by unfolding the clause λ in P w.r.t. an atom A in its body. We assume that the unfolding function *Unfold* is associated with the wqo $\preceq_u$. $Unfold(\lambda, T)$ is defined iff (i) λ is a leaf of T, (ii) the body of λ is non-failing and it contains at least one atom, and (iii) there exists no ancestor α of λ in T such that $bd(\alpha) \preceq_u bd(\lambda)$. Notice that if T consists of the root clause λ only, and λ has non-failing body, then $Unfold(\lambda, T)$ is defined;

(c) $Replace(\Gamma)$ denotes the set of clauses Γ' obtained by applying the constraint replacement rule to each clause γ in Γ as follows:
If γ is of the form $H \leftarrow c, G$ then the clause $H \leftarrow solve(c, X), G$ is in Γ', where X is the set of the linking variables of c in γ.

The *Unfold-Replace* procedure which we now describe, takes as input a set *NewDefs* of definition clauses and, by using the *Unfold* and *Replace* functions defined above, constructs a forest *UForest* of unfolding trees, one for each definition in *NewDefs*. The *Unfold-Replace* procedure is parametric w.r.t. the choice of the unfolding function *Unfold* and its associated well-quasi order $\preceq_u$ on constrained goals.

The Procedure *Unfold-Replace*(*NewDefs*, *UForest*).
Input: a set *NewDefs* of definition clauses.
Output: a forest *UForest* of unfolding trees.

for each clause $\delta \in NewDefs$ **do**
  Let $Utree(\delta)$ be the root clause δ;
  while $Unfold(\lambda, Utree(\delta))$ is defined for some leaf clause λ of $Utree(\delta)$ **do**
    $\Gamma 1 := Unfold(\lambda, Utree(\delta))$;
    $\Gamma 2 := Replace(\Gamma 1)$;
    Expand $Utree(\delta)$ by making every clause in $\Gamma 2$ a son of λ
  end-while
end-for
$UForest := \{Utree(\delta) \mid \delta \in NewDefs\}$ □

Notice that, by the properties of the unfolding function *Unfold*, for each clause δ in *NewDefs* with non-failing body, the tree $Utree(\delta)$ is constructed by applying the unfolding rule at least once.

2.5.2 The Define-Fold Procedure

The *Define-Fold* procedure takes as input a forest *UForest* of unfolding trees, constructed by the *Unfold-Replace* procedure, and a definition tree *Defstree* and it produces as output: (i) a possibly empty set *NewDefs* of new definition clauses, and (ii) a set *FoldedCls* of clauses derived from the leaves of *UForest* which have non-failing bodies, by a (possibly empty) sequence of applications of the constrained atomic folding rule. The definition clauses in *NewDefs*, together with those in *Defstree*, make it possible to fold each leaf of *UForest* with non-failing body w.r.t. each atom occurring in that same leaf. The definition clauses in *NewDefs* are added to the tree *Defstree* as new leaves.

We now introduce the notion of *folding equivalence* between definition clauses. This notion is used to avoid the introduction of unnecessary new definition clauses.

Definition 2.5.1. Given two clauses, δ_1 of the form $H_1 \leftarrow c_1, A_1$ and δ_2 of the form $H_2 \leftarrow c_2, A_2$, we say that δ_1 and δ_2 are *folding equivalent* iff there exists a variable renaming ρ such that (i) $A_1\rho = A_2$, (ii) $\mathcal{D} \models \forall X(c_1\rho \leftrightarrow c_2)$ where $X = FV(c_1\rho, c_2)$, and (iii) $vars(H_1\rho) = vars(H_2)$.

For example, the clauses $new1(X,Y) \leftarrow X > Y,\ p(Y)$ and $new2(V,U) \leftarrow V < U,\ p(V)$ are folding equivalent. We have that, if δ_1 and δ_2 are folding equivalent clauses, then a clause can be folded using δ_1 iff it can be folded using δ_2.

Notice that as consequence of the definitions of the widening operator and the generalization function given in Section 2.4, we have the following property, which will be useful for proving the termination of the contextual specialization strategy.

Property FE: For any clause γ_0 and infinite sequence $\{\delta_i \mid i \in \mathbf{N}\}$ of clauses, if $\{\gamma_i \mid i \in \mathbf{N}\}$ is the infinite sequence of clauses such that, for all $i \in \mathbf{N}$, $\gamma_{i+1} = gen(\gamma_i, \delta_i)$, then there exists an index h such that $\forall k \geq h$, the clauses γ_h and γ_k are folding equivalent.

We now introduce a *definition function Define*, which takes as input a definition tree *Defstree*, a leaf clause δ of *Defstree*, and a call pattern $\langle c, A, Y\rangle$ of a leaf clause λ of $Utree(\delta)$, and produces as output a clause to be used for folding λ w.r.t. A. *Define* is parametric w.r.t. the choices of a wqo $\preceq_g$ and a clause generalization function *gen*.

The Definition Function *Define*$(Defstree, \delta, \langle c, A, Y \rangle)$.

Let η be the clause: $newp(Y) \leftarrow c, A$, where *newp* is a new predicate symbol.
if η is folding equivalent to a clause ϑ in *Defstree*
then return ϑ
else Let π be the path from the root of *Defstree* to clause δ
  if in π there exists a clause of the form $H \leftarrow d, B$ such that
    (1) $(d, B) \preceq_g (c, A)$ and (2) A and B have the same predicate
  then let α be the last clause in π with properties (1) and (2)
    if $gen(\alpha, \eta)$ is folding equivalent to a clause ϑ in *Defstree*
    then return ϑ
    else return $gen(\alpha, \eta)$ (Case G)
  else return η (Case F). □

Now we are ready to present the *Define-Fold* procedure.

The Procedure *Define-Fold*$(UForest, Defstree, NewDefs, FoldedCls)$.
Input: a forest *UForest* of unfolding trees and a definition tree *Defstree*. Each root of the trees in *UForest* is a leaf clause of *Defstree*.
Output: a set *NewDefs* of new definition clauses and a set *FoldedCls* of derived clauses.

NewDefs $:= \emptyset$; *FoldedCls* $:= \emptyset$;
for each unfolding tree *Utree*(δ) in *UForest* **do**
  for each leaf clause λ of *Utree*(δ) with non-failing body **do**
    Let λ be of the form $A_0 \leftarrow d, A_1, \ldots, A_k$, with $k \geq 0$.
    if $k = 0$ **then** *FoldedCls* $:=$ *FoldedCls* $\cup \{\lambda\}$
    else begin
      $\gamma_1 := \lambda$;
      for $i = 1, \ldots, k$ **do**
        Let cp_i be the call pattern of γ_i for A_i and let η be the clause *Define*$(Defstree, \delta, cp_i)$.
        if η is not in *Defstree* **then**
          begin expand *Defstree* by making η a son of δ;
            NewDefs $:=$ *NewDefs* $\cup \{\eta\}$ **end** ;
        Fold γ_i w.r.t. A_i by using η thereby deriving clause γ_{i+1};
      end-for ;
      FoldedCls $:=$ *FoldedCls* $\cup \{\gamma_{k+1}\}$
      end
  end-for
end-for □

Notice that the clauses whose body consists of a satisfiable constraint only, are not folded, and they are added to *FoldedCls*. Moreover the clauses with failing body are not folded, and they are not added to the set *FoldedCls*. This treatment of the clauses with failing body can be viewed as an implicit application of the clause removal rule R4f.

2.5.3 The Contextual Specialization Strategy

We now present our strategy for contextual specialization of CLP($\mathcal{D}$) programs. It consists of two phases. During Phase A we apply the unfolding, constraint replacement, constrained atomic definition, and constrained atomic folding rules, according to the *Unfold-Replace* and *Define-Fold* procedures. During Phase B we eliminate redundant constraints by a suitable application of the contextual constraint replacement rule. During Phase C we apply the rule for removing subsumed clauses, the unfolding rule w.r.t. valid and failed atoms and the rule for removing useless clauses.

Notice that the condition $FV(c) \subseteq X$ on the input to the contextual specialization strategy below, is not actually a restriction, because our constraints are closed w.r.t. existential quantification.

Contextual Specialization Strategy

Input: (i) A CLP($\mathcal{D}$) program P and
(ii) a constrained atom $c, p(X)$ such that $FV(c) \subseteq X$.
Output: A CLP($\mathcal{D}$) program P_s and an atom $p_s(X)$.

Phase A. By the definition rule introduce a clause δ_0 of the form $p_s(X) \leftarrow c, p(X)$. Let *Defstree* consist of clause δ_0 only.
$P_s := \emptyset$; $NewDefs := \{\delta_0\}$;

while $NewDefs \neq \emptyset$ **do**
  Unfold-Replace($NewDefs$, $UForest$);
  Define-Fold($UForest$, $Defstree$, $NewDefs$, $FoldedCls$);
  $P_s := P_s \cup FoldedCls$
end-while

Phase B. [*Contextual Constraint Replacement*]
Let P_s be a program of the form $\{\gamma_1, \ldots, \gamma_p\}$ and
let $\mathbf{C}$ be the set $\{(solve(c, X), p_s(X))\} \cup \{(d, A) \mid \langle d, A, Y\rangle \in CP(P_s)\}$ of constrained atoms.
for $i = 1, \ldots, p$ **do**
  Let γ_i be a clause of the form $q(X) \leftarrow e_1, \ldots, e_n, G$
  where $e_1, \ldots, e_n$ are basic constraints with free variables in $X \cup vars(G)$;
  Let $\mathbf{C}_q$ be the set $\{(d_1, q(X)), \ldots, (d_k, q(X))\}$ of all renamed
  constrained atoms $d, q(X)$ in $\mathbf{C}$;

Let f be the conjunction of all e_j's such that
$\mathcal{D} \models \forall\, (d_r \rightarrow e_j)$ does not hold;
Apply the contextual constraint replacement rule w.r.t. C_q
thereby replacing γ_i by the clause $q(X) \leftarrow f, G$;
endfor

Notice that the use of the contextual constraint replacement rule is justified by the fact that, for all $i = 1, \ldots, k$, $\mathcal{D} \models \forall X\, (d_i \rightarrow (\exists Z\, e \leftrightarrow \exists Z\, f))$, where $Z = vars(G)$.

For reasons of performance, we may replace Condition (iib) above by the following condition:

(iib*) $m \sqsubseteq e_j$, where m denotes the least upper bound $d_1 \sqcup \ldots \sqcup d_k$

which, in general, is stronger than Condition (iib).

Phase C. During this phase we apply the following rules: (i) unfolding, (ii) removal of useless and subsumed clauses, and (iii) constraint replacement. The algorithm for Phase C is as follows.

repeat
  $P' := P_s$;
  Apply to P_s, as long as possible, the rule for removing subsumed clauses;
  Apply to P_s, as long as possible, the unfolding rule
  w.r.t. valid and failed atoms in the current program;
  for all clauses in P_s 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 $P' = P_s$
Remove the useless clauses from P_s; □

2.6 Correctness of the Strategy

Theorem 2.6.1. [Correctness of the Contextual Specialization Strategy] *Let P be a CLP($\mathcal{D}$) program and $c, p(X)$ be a constrained atom with $FV(c) \subseteq X$. Let P_s and $p_s(X)$ be the CLP($\mathcal{D}$) program and the atom obtained by the contextual specialization strategy. Then, for every valuation ν we have that:*

$$\text{if } \mathcal{D} \models \nu(c) \text{ then } \quad \nu(p(X)) \in lm(P, \mathcal{D}) \text{ iff } \nu(p_s(X)) \in lm(P_s, \mathcal{D})$$

Proof. During the application of the contextual specialization strategy, folding is applied only to clauses which have been derived by one or more applications of the unfolding rule, followed by applications of the constraint replacement

rule. Thus, the thesis follows from the correctness of the contextual specialization strategy for general constraint logic programs (see Section 3.5). □

2.7 Termination of the Strategy

We now present the proof of termination of the strategy of Section 2.5.

Definition 2.7.1. The widening operator ∇ *agrees with* the wqo $\preceq$ iff for every atom $q(X)$ and constraints c and d, we have that $c \nabla d, q(X) \preceq c, q(X)$.

We have that any widening operator ∇ agrees with the wqo $\preceq_L$ defined in Example 2.4.2.

Theorem 2.7.2. [Termination of the Contextual Specialization Strategy] *Let P be a CLP($\mathcal{D}$) program, and $c, p(X)$ be a constrained atom with $FV(c) \subseteq X$. If the widening operator ∇ used for clause generalization agrees with the well-quasi order $\preceq_g$, then the contextual specialization strategy terminates.*

Proof. Let us begin by showing the termination of the *Unfold-Replace* procedure. It follows from the properties of the wqo $\preceq_u$ and the hypothesis that for any leaf clause λ of an unfolding tree T, if there exists an ancestor α of λ such that $bd(\alpha) \preceq_u bd(\lambda)$, then $Unfold(\lambda, T)$ is not defined.

Also the *Define-Fold* procedure and Phase B of the contextual specialization strategy trivially terminate because of the absence of while-loop statements in their definitions.

Phase C terminates because during the repeat-loop either (i) we remove clauses by applying the rule for removing subsumed clauses or the unfolding rule w.r.t. a failed atom, or (ii) by applying the unfolding rule w.r.t. a valid atom or the constraint replacement rule, we replace a clause by another clause whose body is strictly smaller.

To prove the termination of the contextual specialization strategy we have to show that the set *NewDefs* of new definitions introduced by the *Define-Fold* procedure will eventually be empty, that is, *Defstree* is a finite tree.

Every node of *Defstree* has finite branching. Indeed, (i) each clause δ occurring in *Defstree* has a number of sons which is not greater than the number of atoms in the bodies of the leaf clauses of $Utree(\delta)$ to be folded, and (ii) for all δ, the unfolding tree $Utree(\delta)$ constructed by the *Unfold-Replace* procedure is finite.

We now show that every path starting from the root of *Defstree* is finite. Consider a generic path π of *Defstree*, of the form $\delta_0 \ldots \delta_k \ldots$, where δ_0 is the root clause of *Defstree*. We can partition the clauses of π into two sets: the set *GenDefs* of clauses which have been introduced as generalizations of one of its ancestors in π (see Case (G) of the definition function *Define*) and the

set *FreshDefs* of all other clauses in π (see Case (F) of the definition function *Define*). In particular, $\delta_0 \in$ *FreshDefs*.

Let us introduce the following binary relation $\preceq_{gp}$ over constrained atoms: $(c, A) \preceq_{gp} (d, B)$ iff (1) $(c, A) \preceq_g (d, B)$ and (2) A and B have the same predicate symbol. We have that $\preceq_{gp}$ is a wqo, because $\preceq_g$ is a wqo and the set of predicate symbols is finite (recall that the predicate symbols occurring in the bodies of the definitions also occur in the initial program). We also have that *gen* agrees with $\preceq_{gp}$ because *gen* agrees with $\preceq_g$.

We will show that for any path π, we can construct a tree $T(\pi)$ whose nodes are the clauses of π, such that:
(Property F*) for any $h > 0$, a clause δ_h is the left son of a clause δ_j iff $\delta_h \in$ *FreshDefs* and δ_j is the last clause in $\delta_0 \ldots \delta_{h-1}$ which is in *FreshDefs*. Thus, $bd(\delta_i) \not\preceq_{gp} bd(\delta_h)$ for all $i = 0, \ldots, h-1$.
(Property G*) for any $h > 0$, a clause δ_h is the right son of a clause δ_j iff $\delta_h \in$ *GenDefs* and δ_h has been introduced as a generalization of δ_j, that is, $\delta_h = gen(\delta_j, \eta)$ for some clause η (which does not belong to the set *Defstree*, and thus, it is not in π). Thus, (i) δ_j is the last clause in $\delta_0 \ldots \delta_{h-1}$ such that $bd(\delta_j) \preceq_{gp} bd(\eta)$, (ii) $bd(\delta_h) \preceq_{gp} bd(\delta_j)$, because *gen* agrees with $\preceq_{gp}$, and (iii) there is no clause in $\delta_0 \ldots \delta_{h-1}$ which is folding equivalent to δ_h.

We will show that for any path π we can construct the tree $T(\pi)$ by proving that, for all finite prefixes $\delta_0 \ldots \delta_k$ of π, there exists $T(\delta_0 \ldots \delta_k)$ satisfying Property F* and Property G* above. The proof proceeds by induction on k.
The base case ($k = 0$) is trivial. For the inductive step, let us assume that Property F* and Property G* hold for $T(\delta_0 \ldots \delta_k)$ and let us show them for $T(\delta_0 \ldots \delta_{k+1})$.

(Case F) Let $\delta_{k+1} \in$ *FreshDefs* and (A) let δ_j be the last clause in $\delta_0 \ldots \delta_k$ such that $\delta_j \in$ *FreshDefs*. Let $T(\delta_0 \ldots \delta_{k+1})$ be the tree obtained from $T(\delta_0 \ldots \delta_k)$ by adding δ_{k+1} as left son of δ_j. Property F* holds for $T(\delta_0 \ldots \delta_{k+1})$: (if part) by construction; (only if part) δ_{k+1} is the only left son of δ_j in $T(\delta_0 \ldots \delta_{k+1})$. Indeed, if there exists a left son δ_h of δ_j, then by inductive hypothesis $\delta_h \in$ *FreshDefs* and $j < h \leq k$, which contradicts the assumption (A). The validity of Property G* for $T(\delta_0 \ldots \delta_{k+1})$ follows immediately from the validity of Property G* for $T(\delta_0 \ldots \delta_k)$.

(Case G) Let $\delta_{k+1} \in$ *GenDefs*, that is, $\delta_{k+1} = gen(\delta_j, \eta)$ for some clause η, where: (B) δ_j is the last clause in $\delta_0 \ldots \delta_k$ such that $bd(\delta_j) \preceq_{gp} bd(\eta)$. Let $T(\delta_0 \ldots \delta_{k+1})$ be the tree obtained from $T(\delta_0 \ldots \delta_k)$ by adding δ_{k+1} as right son of δ_j. The validity of Property F* for $T(\delta_0 \ldots \delta_{k+1})$ follows immediately from the validity of Property F* for $T(\delta_0 \ldots \delta_k)$. Property G* holds for $T(\delta_0 \ldots \delta_{k+1})$ because: (if part) by construction; (only if part) δ_{k+1} is the only right son of δ_j in $T(\delta_0 \ldots \delta_{k+1})$. Indeed, if there exists a right son δ_h of δ_j in $T(\delta_0 \ldots \delta_k)$, then by inductive hypothesis $j < h$ and $bd(\delta_h) \preceq_{gp} bd(\delta_j)$. Thus, by transitivity of

$\preceq_{gp}$ we have $bd(\delta_h) \preceq_{gp} bd(\eta)$ and $j < h$, which contradicts the assumption (B).

We now show that a generic path π of *Defstree* is finite by showing that $T(\pi)$ is finite, that is (a) $T(\pi)$ is finitely branching, and (b) each path in $T(\pi)$ is finite.
(a) By Properties F* and G*, we have that each node of $T(\pi)$ has at most two sons and thus $T(\pi)$ has finite branching.
(b) Consider a path τ from the root of $T(\pi)$ of the form: $\delta_0\gamma_1 \ldots \gamma_k \ldots$ Let γ_h be a clause in τ such that $\gamma_h \in \mathit{GenDefs}$. If such a γ_h does not exist then all clauses in τ belong to *FreshDefs*, and thus, for all distinct $i, j \geq 0$ we have $\gamma_j \not\preceq_{gp} \gamma_i$. In this case τ cannot be infinite because $\preceq_{gp}$ is a wqo. If such a γ_h does exist, then by Properties F* and G* the suffix of τ of the form: $\gamma_h\gamma_{h+1} \ldots$ is such that for all $i \geq h$, $\gamma_{i+1} = gen(\gamma_i, \eta_i)$. By Point (iii) of Property G* we have that: for all $i \geq h$ and for all $j < i$, clause γ_j is not folding equivalent to γ_i. The path $\gamma_h\gamma_{h+1} \ldots$ is finite because, by Property FE of Section 2.5, if it were infinite, then there exist two folding equivalent clauses γ_s and γ_t, with $h \leq s < t$. Thus, τ is finite. □

2.8 An Extended Example

Let us consider the following CLP($\mathcal{R}_{lin}$) program *Mmod*:

1. $mmod(I, J, M) \leftarrow I \geq J,\ M = 0$
2. $mmod(I, J, M) \leftarrow I < J,\ I1 = I+1,\ mod(I, L),\ mmod(I1, J, M1),$
   $M = M1+L$
3. $mod(X, M) \leftarrow X \geq 0,\ M = X$
4. $mod(X, M) \leftarrow X < 0,\ M = -X$

$Mmod(I, J, M)$ holds if and only if $M = |I| + |I+1| + \cdots + |I+k|$ and k is the largest integer such that $I+k$ is smaller than J (recall that I, J, and M are real numbers). Let us assume that we want to specialize the program *Mmod* w.r.t. the constrained atom $I = 0,\ J \geq 0,\ mmod(I, J, M)$.

Recall that in $\mathcal{R}_{lin}$ the least upper bound operation $\sqcup$ is defined by the convex hull construction (see Section 2.4). In this example we instantiate our contextual specialization strategy as follows. (i) The function *solve* is the simplifier of conjunctions of linear equations and inequations over the reals implemented in Holzbaur's clp(q,r) solver [36], (ii) $\mathit{Unfold}(\gamma, \mathit{Utree}(\delta))$ returns the set of clauses obtained by unfolding clause γ w.r.t. the *leftmost* (in the textual order) atom A in its body, (iii) the well-quasi order $\preceq_u$ is the relation $\preceq_L$ over constrained goals (see Section 2.4), (iv) the function *gen* for clause generalization is the one introduced in Example 1 (see Section 2.4), and (v) the wqo $\preceq_g$ is the restriction of $\preceq_L$ to constrained atoms.

We apply the contextual specialization strategy as follows.

Phase A. We start off from the tree *Defstree* which consists of the root clause 5 only, where:

5. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{\geq}0,\ mmod(I, J, M)$

We apply the *Unfold-Replace* procedure as follows. The input for the procedure consists of the set *NewDefs* = {clause 5}. Let *Utree*(clause 5) consist of the root clause 5. The only atom in the body of clause 5 is $mmod(I, J, M)$, and *Unfold*(clause 5, *Utree*(clause 5)) is the set {clause 6, clause 7} where:

6. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{\geq}0,\ I{\geq}J,\ M{=}0$
7. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{\geq}0,\ I{<}J,\ I1{=}I{+}1,\ mod(I, L),$
   $mmod(I1, J, M1),\ M = M1{+}L$

We apply the constraint replacement rule using *solve*, thereby obtaining *Replace*({clause 6, clause 7}) = {clause 8, clause 9} where:

8. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{=}0,\ M{=}0$
9. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{>}0,\ I1{=}1,\ mod(I, L),\ mmod(I1, J, M1),$
   $M = M1{+}L$

Now we expand *Utree*(clause 5) by making clauses 8 and 9 sons of clause 5. *Unfold*(clause 8, *Utree*(clause 5)) is not defined, because the body of clause 8 contains no atom, and thus, clause 8 is not unfolded.
The construction of *Utree*(clause 5) proceeds by first unfolding clause 9 w.r.t. $mod(I, L)$ and then applying the constraint replacement rule. We get the following clauses:

10. $mmod_s(I, J, M) \leftarrow false,\ mmod(I1, J, M1),\ M = M1{+}L$
10.1 $mmod_s(I, J, M) \leftarrow I{=}0,\ J{>}0,\ I1{=}1,\ mmod(I1, J, M)$

We expand *Utree*(clause 5) by making clauses 10 and 10.1 sons of clause 9. There is no leaf clause λ of *Utree*(clause 5) such that *Unfold*(λ, *Utree*(clause 5)) is defined. Indeed: (a) clause 10.1 is the only leaf of *Utree*(clause 5) whose body is non-failing and contains at least one atom, and (b) clause 5 is an ancestor of clause 10.1 such that bd(clause 5) $\preceq_L$ bd(clause 10.1). Thus, the *Unfold-Replace* procedure terminates with output *UForest* = {*Utree*(clause 5)}. The leaves of *Utree*(clause 5) are the clauses 8, 10, and 10.1.

We now apply the *Define-Fold* procedure as follows. The input for the procedure consists of the forest {*Utree*(clause 5)} and the definition tree *Defstree* made out of the root clause 5 only. The leaves of *Utree*(clause 5) with non-failing body are clause 8 and clause 10.1.
The body of clause 8 contains no atom, and so we add clause 8 to *FoldedCls*.
The body of clause 10.1 contains one atom, and the only call pattern of clause 10.1 is $\langle c, A, Y\rangle$, where c is $(J > 0,\ I1 = 1)$, A is $mmod(I1, J, M)$, and Y is $\{I1, J, M\}$.
We now compute *Define*(*Defstree*, clause 5, $\langle c, A, Y\rangle$). We consider the following clause:

η. $newp(I1, J, M) \leftarrow J > 0,\ I1 = 1,\ mmod(I1, J, M)$

Since there is no clause in *Defstree* which is folding equivalent to clause η and $bd(\text{clause } 5) \preceq_L bd(\eta)$, we compute clause $gen(\text{clause } 5, \eta)$, which is (see Example 1 at the end of Section 2.4):

11. $genp(I1, J, M) \leftarrow I1 \geq 0,\ J \geq 0,\ mmod(I1, J, M)$

Since there is no clause in *Defstree* which is folding equivalent to clause 11 we are in Case (G). We expand *Defstree* by making clause 11 a son of clause 5 and we add clause 11 to the set *NewDefs*. Then we fold clause 10.1 by using the definition clause 11 and we get:

12. $mmod_s(I, J, M) \leftarrow I = 0,\ J > 0,\ I1 = 1,\ genp(I1, J, M)$

The *Define-Fold* procedure terminates with output *FoldedCls* = {clause 8, clause 12} and *NewDefs* = {clause 11}. Since *NewDefs* is non-empty, we continue the execution of the while-loop of the contextual specialization strategy. We apply the *Unfold-Replace* procedure as follows. The input for the procedure consists of the set *NewDefs* = {clause 11}. At the beginning, *Utree*(clause 11) consists of the root clause 11. The only atom in the body of clause 11 is $mmod(I1, J, M)$, and *Unfold*(clause 11, *Utree*(clause 11)) is the set {clause 13, clause 14} where:

13. $genp(I1, J, M) \leftarrow I1 \geq 0,\ J \geq 0,\ I1 \geq J,\ M = 0$
14. $genp(I1, J, M) \leftarrow I1 \geq 0,\ J \geq 0,\ I1 < J,\ I2 = I1 + 1,\ mod(I1, L),$
$mmod(I2, J, M2),\ M = M2 + L$

We apply the constraint replacement rule using the *solve* function, and we get *Replace*({clause 13, clause 14}) = {clause 15, clause 15.1} where:

15. $genp(I1, J, M) \leftarrow I1 \geq J,\ J \geq 0,\ M = 0$
15.1 $genp(I1, J, M) \leftarrow I1 \geq 0,\ I1 < J,\ I2 = I1 + 1,\ mod(I1, L),$
$mmod(I2, J, M2),\ M = M2 + L$

Now we expand *Utree*(clause 11)) by making clauses 15 and 15.1 sons of clause 11. *Unfold*(clause 15, *Utree*(clause 11)) is not defined, because the body of clause 15 contains no atom.

The construction of *Utree*(clause 11) proceeds by first unfolding clause 15.1 w.r.t. $mod(I1, L)$ and then applying the constraint replacement rule. We get, as the reader may verify, the following clauses:

16 $genp(I1, J, M) \leftarrow false,\ mmod(I2, J, M2)$
16.1 $genp(I1, J, M) \leftarrow I1 \geq 0,\ I1 < J,\ I2 = I1 + 1,$
$mmod(I2, J, M2),\ M = M2 + I1$

We expand *Utree*(clause 11) by making clauses 16 and 16.1 sons of clause 15.1. There is no leaf clause λ of the unfolding tree *Utree*(clause 11) such that *Unfold*(λ, *Utree*(clause 11)) is defined. Indeed: (a) clause 16.1 is the only leaf of *Utree*(clause 11) whose body is non-failing and it contains at least one atom, and (b) clause 11 is an ancestor of clause 16.1 such that $bd(\text{clause } 11) \preceq_L$

bd(clause 16.1). Thus, the *Unfold-Replace* procedure terminates with output $UForest = \{Utree(\text{clause } 11)\}$.

We now apply the *Define-Fold* procedure as follows. The input for the procedure consists of the forest $\{Utree(\text{clause } 11)\}$ and the definition tree *Defstree* made out of the root clause 5 and its son clause 11. The leaf clauses of $Utree(\text{clause } 11)$ with non-failing body are clause 15 and clause 16.1.
The body of clause 15 contains no atom, and so we add clause 15 to *FoldedCls*.
The body of clause 16.1 contains one atom and $\langle c', A', Y' \rangle$ is the only call pattern of clause 16.1 where c' is $(I2 \geq 1,\ J > I2 - 1)$, A' is $mmod(I2, J, M2)$, and Y' is $\{I2, J, M2\}$.
We now compute $Define(Defstree, \text{clause } 11, \langle c', A', Y' \rangle)$. Let us consider the clause:

$\eta 1.\quad newq(I2, J, M2) \leftarrow I2 \geq 1,\ J > I2 - 1,\ mmod(I2, J, M2)$

Since there is no clause in *Defstree* which is folding equivalent to $\eta 1$ and $bd(\text{clause } 11) \preceq_L bd(\eta 1)$, we compute clause $gen(\text{clause } 11, \eta 1)$, which is clause 11 itself. Then we fold clause 16.1 by using the definition clause 11 and we get:

$$17.\ \ genp(I1, J, M) \leftarrow I1 \geq 0,\ I1 < J,\ I2 = I1 + 1,\ genp(I2, J, M2),\ M = M2 + I1$$

The *Define-Fold* procedure terminates with output $FoldedCls = \{\text{clause } 15, \text{clause } 17\}$ and $NewDefs = \emptyset$.

Since during the last application of the *Define-Fold* procedure we did not introduce any new definition, the while-loop of the contextual specialization strategy terminates and we get program P_s made out of clauses 8, 12, 15, and 17.

Phase B. By computing the call patterns of P_s and performing a variable renaming we get:

$$\mathbf{C} = \{(I = 0,\ J \geq 0,\ mmod_s(I, J, M)),\ (I = 1,\ J > 0,\ genp(I, J, M)),\ (I \geq 1,\ J > I - 1,\ genp(I, J, M))\}$$

We then process clauses 8, 12, 15, and 17 as indicated in the contextual specialization strategy where we use Condition (iib*) instead of Condition (iib). We compute the least upper bounds:
for $mmod_s$: $m_1 = (I = 0,\ J \geq 0)$ and for $genp$: $m_2 = (I \geq 1,\ J > I - 1)$.
For clause 8, since $m_1 \sqsubseteq I = 0$, we get:

$8.c\quad mmod_s(I, J, M) \leftarrow J = 0,\ M = 0$

For clause 12, since $m_1 \sqsubseteq I = 0$, we get:

$12.c\quad mmod_s(I, J, M) \leftarrow J > 0,\ I1 = 1,\ genp(I1, J, M)$

For clause 15, since $m_2 \sqsubseteq J \geq 0$, after a variable renaming, we get:

15.c $genp(I, J, M) \leftarrow I \geq J,\ M = 0$

For clause 17, since $m_2 \sqsubseteq I \geq 0$, after a variable renaming, we get:

17.c $genp(I, J, M) \leftarrow I < J,\ I2 = I+1,\ genp(I2, J, M2),\ M = M2+I$

The output of Phase B is the program consisting of clauses 8.c, 12.c, 15.c, and 17.c.

Phase C. The current program contains no facts and no valid or failed atoms. Thus, Phase C leaves the program unchanged.

The final program P_s we have derived, consists of clauses 8.c, 12.c, 15.c, and 17.c.

2.9 Experimental Results

The following table shows the speedups achieved by applying our constraint specialization strategy to some CLP programs. The speedups after Phase A and after Phase C are both computed w.r.t. the initial program. The Dynamic Input Size denotes the size of the constrained goal which is supplied to the specialized program. The experimental results were obtained by using SICStus Prolog 3.8.5 [37] and the clp(q,r) solver [36].

Program	**Dynamic Input Size**	**Speedup after Phase A**	**Speedup after Phase C**
Mmod	$\|J\| = 250$	3.26	3.78
Mmod	$\|J\| = 25000$	345	388
Summatch (†)	$\|S\| = 500$	451	915
Summatch (†)	$\|S\| = 1000$	881	1788
Summatch (††)	$\|S\| = 500$	818	2159
Summatch (††)	$\|S\| = 1000$	1594	4225
Cryptosum	—	1.27	1.27

- *Mmod* is the program described in Section 2.8 which defines the predicate *mmod*(*I*,*J*,*M*). It has been specialized w.r.t. $I = 0, J \geq 0$.
- *Summatch* is a program which defines a predicate *summatch*(*P*,*S*) which holds iff there exists a substring G of S such that: (i) G and P have the same length, and (ii) the sum of the elements of G is equal to the sum of the elements of P. It has been specialized w.r.t. (†) a list P of 3 nonnegative integers whose sum is at most 5, and (††) a list P of 10 nonnegative integers whose sum is at most 5.
- *Cryptosum* is a program which solves a cryptoarithmetic puzzle over three lists $L1$, $L2$, and $L3$ of digits, such that $L1+L2=L3$. It has been specialized w.r.t. SEND+MORE=MONEY.

The specialized programs were derived automatically by using the MAP transformation system, which provides support both for interactive derivations and for automatic specializations based on the parameterized strategy presented in Section 2.5. The MAP system is described in Appendix A. The source code for the examples can be found in Appendix B.

2.10 Related Work

There are various methods for specializing programs w.r.t. properties of their input data [10, 16, 53, 61, 66]. Contextual specialization can be viewed as one of these methods. In this chapter we have presented a set of transformation rules and an automated strategy for the contextual specialization of constraint logic programs over a domain $\mathcal{D}$.

For our specialization strategy we have assumed the existence of a solver which simplifies constraints in a given domain $\mathcal{D}$. We abstractly represent that solver as a computable total function *solve* and we do not make any assumption on how this function is computed. The motivation for this choice comes from the observation that most of the available constraint solvers are based on the so called *black box* approach and they provide the user with very limited ways of interaction. In particular, in the black box approach, it is not possible to control the constraint solving process.

For this reason the efficiency improvements which can be achieved by our program specialization strategy are limited to the way the constraints are generated and interact with each other. Indeed, at specialization time we may simplify constraints by discovering inconsistencies, exploiting entailment, and applying the function *solve*. These opportunities for constraint simplification are triggered by applications of the unfolding rule and are realized by the constraint replacement rule. The unfolding rule, in fact, may gather in a single clause constraints which come from different clauses.

However, our specialization strategy cannot improve the efficiency of the constraint solving algorithms. To overcome this limitation we could extend our method to programs based on *glass box* solvers, for example by using Constraint Handling Rules [31], a high-level language designed for the purpose of building application-specific constraint solvers.

In our specialization strategy we have also assumed the existence of: (i) a function *Unfold* and a well-quasi order $\preceq_u$ between constrained goals for guiding the unfolding process, (ii) a clause generalization function *gen* parameterized by a widening operator ∇, and (iii) a well-quasi order $\preceq_g$ between constrained atoms, which activates the clause generalization process during program specialization. We have shown that our specialization strategy preserves the least $\mathcal{D}$-model and it terminates.

The hypothesis that the function *solve* is complete w.r.t. satisfiability, makes our approach different from the one in [10], where program specialization is based on some undecidable properties and thus, it cannot be easily automated.

Partial evaluation of logic programs [50], also called *partial deduction*, is the technique for program specialization which is most related to ours. However, we would like to mention the following differences which make our contextual specialization technique a proper extension of the traditional techniques for partial evaluation.
(1) The most apparent difference is that traditional techniques for partial evaluation do not handle constraints, so that our optimizations concerning constraint solving cannot be performed.
(2) Partial evaluation may require post processing methods, like Redundant Argument Filtering [19], to minimize the number of variables occurring in clauses, whereas we use the folding rule during program specialization to avoid redundant occurrences of variables.
(3) The use of our contextual constraint replacement rule R5 allows us to perform optimizations which cannot be performed by applying the techniques for partial evaluation of logic programs presented in [48, 84]. Indeed, our contextual constraint replacement rule takes into account, for simplifying the clauses of a predicate, say p, the set of constraints which occur in the clauses containing a call of p. Nor can contextual constraint replacement be performed by using the transformation rules presented in [9, 25, 51].
(4) What it is usually done by automatic partial evaluation techniques (see, for instance, [46]) basically corresponds with Phase A of our contextual specialization strategy. Similarly to partial evaluation, in that phase we address *local control* and *global control* issues. In our approach local control refers to the termination of the *Unfold-Replace* procedure, while global control refers to the termination of Phase A of the contextual specialization strategy as a whole. In particular, global control refers to the policy of introducing new constrained atoms which generalize old constrained atoms. For these control issues we extend to the case of constrained logic programs the well-quasi order techniques used for proving termination in the field of rewriting systems [21]. These techniques were also used for partial evaluation [44, 46]. With regard to these control issues the main difference between partial evaluation and our approach is that generalization used in partial evaluation can be seen as a particular instance of our generalization by taking both the least upper bound operator and the widening operator to be the most specific generalization over finite terms.

Among the many techniques for simplifying and manipulating constraints to get more efficient specialized programs, here we want to mention the following methods which are related to ours.

In [66] the authors propose a method based on abstract interpretation, for the implementation of multiple specialization of logic programs. Particular emphasis is given to program parallelization. Similarly to their work, our specialization strategy may produce several specialized versions of the same predicate by introducing different definitions corresponding to different call patterns.

In [53] the authors present a methodology for compiling CLP($\mathcal{D}$) languages so that the workload of the constraint solver is reduced. However, their methodology may generate *non-monotonic* CLP($\mathcal{D}$) programs, whose execution requires the ability of removing constraints from the constraint store. In contrast, in our approach we generate monotonic CLP($\mathcal{D}$) programs which at runtime do not remove constraints from the store. However, we require the solver to test for the satisfiability and entailment of constraints, and to compute the existential closure of constraints w.r.t. given sets of variables. These capabilities are indeed provided by most constraint solvers.

Finally, our work is related to the approach presented in [16] for a strict first-order functional language. As in that paper, we too specify the context of use w.r.t. which the programs should be specialized, by providing a property which may cover an infinite set of queries.

Chapter 3

Specialization of General Constraint Logic Programs

In this chapter we extend the framework presented in Chapter 2 so as to perform specialization of constraint logic programs with negation. This extension will also be used for the verification of temporal properties of concurrent systems as described in Chapter 4.

We will consider the class of locally stratified constraint logic programs. This class is interesting because it is expressive enough to contain many of the constraint logic programs which are used in practice. Moreover, all major approaches to the semantics of negation coincide for locally stratified CLP programs. In particular, given a locally stratified constraint logic program P, the unique perfect model [65] of P is equal to the unique stable model [34] of P and to the well-founded model [83] of P. The perfect model of a locally stratified constraint logic program is constructed in a way which is very similar to the case of locally stratified logic programs and most approaches for dealing with negation can be extended from logic programming to constraint logic programming.

Contextual specialization for general constraint logic programs is defined as follows. Given a locally stratified CLP($\mathcal{D}$) program P and a constrained atom c, A derive a locally stratified program P_s and an atom A_s such that, for every valuation ν we have that:

($Contextual\ Specialization$)

$$\text{if } \mathcal{D} \models \nu(c) \text{ then } \quad \nu(A) \in M(P) \text{ iff } \nu(A_s) \in M(P_s)$$

where $M(P)$ denotes the perfect model of P.

We now introduce some basic notions and notational conventions on constraint logic programs with negation. This is an extension of the material presented in 2.1. For notions on logic programming with negation which are not defined here the reader may refer to [6].

3.1 Constraint Logic Programming with Negation

A *negated atom* is a formula of the form $\neg A$ where A is an atom. A *literal* is either an atom A, also called *positive literal*, or a negated atom $\neg A$, also called *negative literal*. Literals are denoted by L, possibly with subscripts. A *constrained literal* is the conjunction of a constraint and a literal. A *goal* is a (possibly empty) conjunction of literals, and it is denoted by G, possibly with subscripts. *Clauses* are of the form $H \leftarrow c, L_1, \ldots, L_n$. A *general constraint logic program*, simply called *program* in the following chapters, is a finite set of clauses.

We now extend the domain of a valuation ν to negative literals as follows: if L is the negated atom $\neg A$, then $\nu(L)$ is the ground literal $\neg\nu(A)$. Given a program P, we define $ground(P)$ as the following set of ground clauses:

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

Given a $\mathcal{D}$-interpretation I and a constraint-free, ground clause γ of the form $H \leftarrow L_1, \ldots, L_m$, 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 $A_i \in I$.

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 [6, 65] and are parametric w.r.t. the interpretation of the constraints [38, 39].

A *local stratification* is a function σ: $\mathcal{B}_\mathcal{D} \rightarrow W$, 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 [6, 65], 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 $A \in M_\tau$.

Notice that the construction of the perfect model of a program presented above is different from the construction of the perfect model presented in [65]. However, as the reader may verify, in the case of locally stratified programs the two constructions yield the same model.

Similarly to the case of logic programs [6, 65], we have the following result.

Theorem 3. *Every locally stratified constraint logic program has a unique perfect model.*

As already mentioned, a program P with locally stratified negation has a unique perfect model $M(P)$ which coincides with its unique stable model, and its total well-founded model.

3.2 Rules for Transforming General Constraint Logic Programs

The following rules are an extension of the rules presented in Section 2.2 for definite CLP programs to the case of CLP programs with locally stratified negation. We want to point out that some of the rules presented here are in fact identical to rules presented in Section 2.2, except for the fact that they refer to general programs instead of definite programs. In particular, the *constrained atomic definition* rule R1, the *positive unfolding* rule R2p, Case (P) of the *constrained atomic folding* rule R3, the *clause removal* rules R4f, R4s and R4u, and the *constraint replacement* rule R5r presented in this section are identical to rules R1, R2, R3, R4f, R4s, R4u and R5r, respectively, which

have been presented in Section 2.2. We reproduce them below for the reader's convenience. The *negative unfolding* rule R2n, Case (N) of the *constrained atomic folding* rule R3 and the *contextual constraint replacement* rule R5n are proper extensions of the rules for definite programs to general programs.

R1. Constrained Atomic Definition. By *constrained atomic definition* (or *definition*, for short), we introduce the new clause

$$\delta: \ \ newp(X) \leftarrow c, A$$

which is said to be a *definition*, where: (i) *newp* is a predicate symbol not occurring in $P_0, \ldots, P_k$, (ii) X is a sequence of distinct variables occurring in the constrained atom c, A, and (iii) the predicate symbol of A occurs in P_0. From program P_k we derive the new program P_{k+1} which is $P_k \cup \{\delta\}$.
For $i \geq 0$, $Defs_i$ is the set of definitions introduced during the transformation sequence $P_0, \ldots, P_i$. In particular, $Defs_0 = \emptyset$.

R2p. Positive Unfolding. Let $\gamma: \ H \leftarrow c, G_L, A, G_R$ be a renamed apart clause of P_k and let $\{A_j \leftarrow c_j, G_j \mid j = 1, \ldots, m\}$ be the set of *all* clauses in P_k such that the atoms A and A_j have the same predicate symbol. For $j = 1, \ldots, m$, let us consider the clause

$$\gamma_j: \ \ H \leftarrow c,\ A{=}A_j,\ c_j,\ G_L, G_j, G_R$$

where $A{=}A_j$ stands for the conjunction of the equalities between the corresponding arguments. Then, by *unfolding* clause γ w.r.t. atom A, from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_j \mid j = 1, \ldots, m\}$.

R2n. Negative Unfolding. Let $\gamma: \ H \leftarrow c, G_L, \neg A, G_R$ be a renamed apart clause of P_k. The negative unfolding rule can be applied in the following two cases.

(Case F) If A is failed in program P_k then let γ_1 be the clause $H \leftarrow c, G_L, G_R$. By *unfolding* clause γ w.r.t. the negated atom $\neg A$, from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_1\}$.

(Case V) If A is valid in program P_k then, by *unfolding* clause γ w.r.t. the negated atom $\neg A$, from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R3. Constrained Atomic Folding. Let $\gamma: \ A \leftarrow c, G_L, L, G_R$ be a clause of P_k where literal L is either the atom B or the negated atom $\neg B$. Let $\delta: \ newp(X) \leftarrow d, B$ be a variant of a clause in $Defs_k$. For the application of the constrained atomic folding rule we can distinguish the following two cases which depend on the form of the literal L.

(Case P) Literal L is the atom B .
Suppose that: (i) $\mathcal{D} \models \forall Y\ (c \rightarrow d)$, where $Y = FV(c, d)$, and (ii) no variable in $FV(\delta) - X$ occurs in $FV(A, c, G_L, G_R)$. Then, by *folding* clause γ w.r.t. L using δ, we derive the new clause

$$\gamma_f: \ A \leftarrow c, G_L, newp(X), G_R$$

(Case N) Literal L is the negated atom $\neg B$.
Suppose that: (i) $\mathcal{D} \models \forall Y\,(c \rightarrow d)$, where $Y = FV(c,d)$, and (ii) for each variable Z in $FV(\delta) - X$ there exists $t \in D$ such that $\mathcal{D} \models \forall W\,(d \rightarrow Z = t)$, where $W = FV(d)$. Then, by *folding* clause γ w.r.t. L using δ, we derive the new clause

$$\gamma_f: \ A \leftarrow c, G_L, \neg newp(X), G_R$$

In both cases, from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_f\}$.

In the following, we will also refer to Case (P) and Case (N) of rule R3 as rule R3p and rule R3n, respectively.

R4f. Clause Removal: Unsatisfiable Body. Let $\gamma: \ A \leftarrow c, G$ be a clause of P_k. If the constraint c is *unsatisfiable*, that is, $solve(c, \emptyset) = false$, then from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R4s. Clause Removal: Subsumed Clause. Let $\gamma: \ p(X) \leftarrow c, G$ with $(c, G) \neq true$, be a clause of P_k. If P_k contains a fact λ of the form $p(Y) \leftarrow$ then from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R4u. Clause Removal: Useless Clauses. Let Γ be the set of useless clauses in P_k. Then, by *removing useless clauses* from program P_k we derive the new program P_{k+1} which is $P_k - \Gamma$.

R5n. Contextual Constraint Replacement. Let $\mathbf{C}$ be a set of constrained atoms. Let γ be a renamed apart clause in P_k of the form: $p(U) \leftarrow c_1, G$. Suppose that for some constraint c_2, and for every constrained atom $c, p(V)$ in $\mathbf{C}$, we have that

$$\mathcal{D} \models \forall X\,((c, U{=}V) \ \rightarrow \ (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2))$$

where: (i) $Y = FV(c_1) - vars(U, G)$, (ii) $Z = FV(c_2) - vars(U, G)$, and (iii) $X = FV(c, U = V, c_1, c_2) - (Y \cup Z)$. Then, we derive program P_{k+1} from program P_k by replacing clause γ by the clause: $p(U) \leftarrow c_2, G$. In this case we say that P_{k+1} has been derived from P_k by contextual constraint replacement w.r.t. $\mathbf{C}$.

The following rule is an instance of rule R5n for $\mathbf{C} = \{true, p(U)\}$.

R5r. Constraint Replacement. Let $\gamma: \ A \leftarrow c_1, G$ be a renamed apart clause of P_k. Assume that $\mathcal{D} \models \forall X(\exists Y\, c_1 \leftrightarrow \exists Z\, c_2)$ where: (i) $Y = FV(c_1) - vars(A, G)$, (ii) $Z = FV(c_2) - vars(A, G)$, and (iii) $X = FV(c_1, c_2) - (Y \cup Z)$. Then from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{A \leftarrow c_2, G\}$.

3.3 Correctness of the Transformation Rules

In this section we study the correctness w.r.t. the perfect model of the transformation rules for general constraint logic programs presented in Section 3.2.

We show that for any transformation sequence $P_0, \ldots, P_n$ where (i) each definition clause is unfolded w.r.t. the atom in its body and (ii) the contextual constraint replacement rule R5n is only applied in its restricted form R5r, we have that the perfect model is preserved, in the sense that, $M(P_0 \cup \mathit{Defs}_n) = \ldots = M(P_n)$ where Defs_n denotes the set of definition clauses introduced during the construction of the transformation sequence.

Without loss of generality, we may assume that the transformation sequence is of the form $P_0, \ldots, P_0 \cup \mathit{Defs}_n, \ldots, P_j, \ldots, P_n$ where (i) Defs_n is the set of definition clauses introduced during the construction of the transformation sequence, (ii) P_j is derived from $P_0 \cup \mathit{Defs}_n$ by unfolding each clause in Defs_n w.r.t. the atom in its body, and (iii) for all $i = j, \ldots, n-1$, program P_{i+1} is not derived from program P_i by an application of the constrained atomic definition rule.

We now introduce some preliminary definitions which will be used in the proofs.

Definition 3.3.1. A *proof tree* for a ground atom A in a CLP program P is a finite tree T such that: (i) the root of T is A, (ii) every internal node B of T is a ground atom (iii) every leaf node of T is either the symbol *true* or a negated ground atom $\neg B$ such that there is no proof tree for B in P, (iv) if an internal node B of T has children $L_1, \ldots, L_k$ then $B \leftarrow L_1, \ldots, L_k$ is a clause in $\mathit{ground}(P)$.
The *size* of a proof tree T is the number $\mathit{size}(T)$ of internal nodes of T.
The *weight* of a ground atom A is

$$\mu(A) = \min\{\mathit{size}(T) \mid T \text{ is a proof tree for } A \text{ in } P_j\}$$

A proof tree T is *weight-consistent* iff for all ground atoms A and B, if B is a child of A in T then $\mu(B) < \mu(A)$.

The proof of correctness proceeds as follows:

1. We show that the transformation rules preserve local stratification. Recall that the perfect model is defined for locally stratified programs.

2. We consider the transformation sequence $P_j, \ldots, P_n$ and we show that for all $i = j, \ldots, n$ and for every ground atom A there exists a proof tree for A in P_i iff there exists a proof tree for A in $P_0 \cup \mathit{Defs}_n$.

3. We establish a correspondence between proof trees in program P and the perfect model of P.

Theorem 3.3.2. [Preservation of Local Stratification]. *Let P_0 be a locally stratified program and let $P_j, \ldots, P_n$ be a transformation sequence which is obtained by applying the transformation rules of Section 3.2 with the condition that the contextual constraint replacement rule R5n is only applied in its restricted form R5r. Then, there exists a function* stratum *such that for all $i = 0, \ldots, n$ program P_i is locally stratified w.r.t.* stratum.

Proof. Let P_0 be locally stratified w.r.t. the function *stratum'*. Consider the function *stratum* which is defined as follows. Let A be a ground atom in the base $\mathcal{B}_{\mathcal{D}}$ of $P_0 \cup \mathit{Defs}_n$.

if the predicate symbol of A occurs in P_0 **then** $\mathit{stratum}(A) = \mathit{stratum}'(A)$
else if there exists a clause δ in $\mathit{ground}(\mathit{Defs}_n)$ such that $A = \mathit{hd}(\delta)$
then $\mathit{stratum}(A) = \max\{\mathit{stratum}'(B) \mid B \in \mathit{bd}(\delta)\}$.

The proof proceeds by induction on i.
Base case ($i = 0$). Trivial.
Induction step. We assume that for all $j \leq i$, P_j is locally stratified w.r.t. *stratum* and we show that P_{i+1} is locally stratified w.r.t. the same function. We consider a clause $\gamma_\nu \in \mathit{ground}(\{\gamma\})$ where $\gamma \in P_{i+1}$, and ν is a valuation such that $\gamma_\nu = \nu(\gamma)$. We proceed by cases.

Case 1. Clause γ is inherited from P_i. Trivial.

Case 2. Clause γ is obtained by constrained atomic definition (rule R1). Straightforward from the definition of the function *stratum*.

Case 3. Clause γ is obtained by positive unfolding (rule R2p). We assume, with no loss of generality, that there exist (renamed apart) clauses α and β in P_i of the form $H \leftarrow c, G_L, A, G_R$ and $B \leftarrow d, G$, respectively. Let clause γ be of the form $H \leftarrow c, d, A = B, G_L, G, G_R$. Since $\gamma_\nu \in \mathit{ground}(P_{i+1})$ we have that $\mathcal{D} \models \nu(c \wedge d \wedge A = B)$ and thus, $\alpha_\nu \in \mathit{ground}(P_i)$ and $\beta_\nu \in \mathit{ground}(P_i)$. By induction hypothesis we have that α_ν and β_ν are locally stratified w.r.t. *stratum*. Thus, clause γ_ν is locally stratified w.r.t. *stratum* because $\mathit{stratum}(\nu(A)) = \mathit{stratum}(\nu(B))$.

Case 4. Clause γ is obtained by negative unfolding (rule R2n - Case F). Let α be the clause in P_i which has been replaced by clause γ. Let γ_ν be of the form $H \leftarrow G_L, G_R$. Thus, there exists a clause β in $\mathit{ground}(\{\alpha\})$ of the form $H \leftarrow G_L, \neg B, G_R$. By induction hypothesis we have that β is locally stratified w.r.t. *stratum*. Thus, so is γ_ν.

Case 5. Clause γ is obtained by folding (rule R3n). Let γ be obtained by folding a clause α in P_i of the form $H \leftarrow c, G_L, L, G_R$ by using a (variant of a) definition clause $\delta \in \mathit{Defs}_n$ of the form $\mathit{newp}(X) \leftarrow d, B$.

(Case P) Literal L is the atom B and clause γ is of the form $H \leftarrow c, G_L, newp(X), G_R$.

Thus, γ_ν is of the form $H' \leftarrow G'_L, newp(t), G'_R$. By the conditions on the applicability of case P of rule R3n we have: (i) $\mathcal{D} \models \forall Y\,(c \rightarrow d)$, where $Y = FV(c,d)$, and (ii) no variable in $FV(\delta) - X$ occurs in $FV(H, c, G_L, G_R)$. Condition (i) ensures that there exists a clause $newp(t) \leftarrow B'$ in $ground(\{\delta\})$ such that $stratum(newp(t)) = stratum(B')$. By Condition (ii) we have that in α there is no constraint on $vars(B) - X$, and thus clause $H' \leftarrow G'_L, B', G'_R$ is in $ground(P_i)$. Thus, by induction hypothesis we have that $stratum(H') \geq stratum(B') = stratum(newp(t))$.
(Case N) Literal L is the negated atom $\neg B$ and clause γ is of the form $H \leftarrow c, G_L, \neg newp(X), G_R$.

By the conditions on the applicability of case N of rule R3n we have: (i) $\mathcal{D} \models \forall Y\,(c \rightarrow d)$, where $Y = FV(c,d)$, and (ii) for each variable Z in $FV(\delta) – X$ there exists $s \in D$ such that $\mathcal{D} \models \forall W\,(d \rightarrow Z = s)$, where $W = FV(d)$. Condition (i) ensures that there exists a clause $newp(t) \leftarrow B'$ in $ground(\{\delta\})$ such that $stratum(newp(t)) = stratum(B')$. By Conditions (i) and (ii) we have that for each variable Z in $FV(\delta) - X$ there exists $s \in D$ such that $\mathcal{D} \models \forall Y\,(c \rightarrow Z = s)$ where $Y = FV(c)$. Thus, clause $H' \leftarrow G'_L, \neg B', G'_R$ is in $ground(P_i)$ and by induction hypothesis we have that $stratum(H') > stratum(B') = stratum(newp(t))$.

Case 6. Clause γ is obtained by constraint replacement (rule R5r) from clause α in P_i. In this case the thesis follows from the inductive hypothesis because $ground(\{\gamma\}) = ground(\{\alpha\})$. □

Proposition 3.3.3. *Let P_0 be a locally stratified CLP program and let $P_0 \cup Defs_n, \ldots, P_j$ be a transformation sequence which is obtained by unfolding each clause in $Defs_n$ w.r.t. the atom in its body. Then, for any ground atom H we have that: there exists a proof tree for H in $P_0 \cup Defs_n$ iff there exists a proof tree for H in P_j.*

Proof. Let *stratum* be a stratification function for $P_0 \cup Defs_n$. By Theorem 3.3.2, all programs in the transformation sequence $P_0 \cup Defs_n, \ldots, P_j$ are locally stratified w.r.t. *stratum*.

In the proof of soundness (respectively, completeness), given a proof tree T for H in P_j (respectively, in $P_0 \cup Defs_n$) we construct a proof tree T' for H in $P_0 \cup Defs_n$ (respectively, in P_j) by well-founded induction on the lexicographic product of the well-founded order over *stratum* and the well-founded order over *size*.

Let $\gamma_\nu \in ground(\{\gamma\})$ be the ground clause of the form $H \leftarrow L_1, \ldots, L_k$ used at the root of T, where γ is a clause in P_j (respectively, in $P_0 \cup Defs_n$) and ν is a valuation such that $\gamma_\nu = \nu(\gamma)$. The inductive hypotheses are:

(*IHstratum*) For all ground atoms A, if $stratum(A) < stratum(H)$ then A has a proof tree in P_j iff A has a proof tree in $P_0 \cup Defs_n$.
(*IHsize*) For all ground atoms A, if $stratum(A) \leq stratum(H)$ and A has a proof tree T_A in P_j (respectively, in $P_0 \cup Defs_n$) such that $size(T_A) < size(T)$, then A has a proof tree T'_A in $P_0 \cup Defs_n$ (respectively, in P_j).

We have the following property which holds for all nodes L of T.
(*Property* 1). If L is an atom then let T_L be the subtree of T rooted at L. We have that T_L is a proof tree for L in P_j (respectively, in $P_0 \cup Defs_n$) and $size(T_L) < size(T)$. Thus, by hypothesis (IHsize) we have that there exists a proof tree for L in $P_0 \cup Defs_n$ (respectively, in P_j).
Otherwise, let L be a negated atom $\neg B$. Since T is a proof tree for H in P_j (respectively, in $P_0 \cup Defs_n$) we have that there is no proof tree for B in P_j (respectively, in $P_0 \cup Defs_n$). Moreover, since program P_j (respectively, $P_0 \cup Defs_n$) is locally stratified w.r.t. *stratum* we have $stratum(B) < stratum(H)$. Thus, we can apply hypothesis (IHstratum) and we have that there is no proof tree for B in $P_0 \cup Defs_n$ (respectively, in P_j).

Now we prove separately the soundness and completeness part.

(*Soundness*) For any ground atom H, if there exists a proof tree for H in P_j then there exists a proof tree for H in $P_0 \cup Defs_n$. We proceed by cases.

Case 1. Clause $\gamma \in P_0 \cup Defs_n$. We construct T' as follows: we use γ_ν at the root, and for all $h = 1, \ldots, k$ such that L_h is an atom A we use T'_A as subtree of T' at A. By Property (1) we have that T' is a proof tree for H in $P_0 \cup Defs_n$.

Case 2. Clause γ is obtained by positive unfolding (rule R2p). Thus, there exist: a (renamed apart) clause α in $Defs_n$ of the form $H \leftarrow c, A$ and a (renamed apart) clause β in P_0 of the form $B \leftarrow d, G$ such that (i) A and B have the same predicate symbol, and (ii) clause γ is of the form $H \leftarrow c, d, A = B, G$. Since $\gamma_\nu \in ground(\{\gamma\})$ we have that $\mathcal{D} \models \nu(c \wedge d \wedge A = B)$ and thus, $\alpha_\nu \in ground(P_0 \cup Defs_n)$ and $\beta_\nu \in ground(P_0 \cup Defs_n)$.
We construct T' as follows: we use α_ν at the root, then we use β_ν at A. The leaves of the current proof tree are $L_1, \ldots, L_k$. We complete the construction of T' by using T'_A as a subtree of T' at L_h, for all $h = 1, \ldots, k$ such that L_h is an atom A. By Property (1) we have that T' is a proof tree for H in $P_0 \cup Defs_n$.

(*Completeness*) For any ground atom H, if there exists a proof tree for H in $P_0 \cup Defs_n$ then there exists a proof tree for H in P_j. We proceed by cases.

Case 1. Clause $\gamma \in P_j$. We construct T' as follows: we use γ_σ at the root, and for all $h = 1, \ldots, k$ such that L_h is an atom A, we use T'_A as a subtree of T' at A. By Property (1) we have that T' is a proof tree for H in P_j.

Case 2. Clause $\gamma \in Defs_n$ is removed by positive unfolding (rule R2p). Let γ be of the form $H' \leftarrow c, A'$. Let $R = \{A_k \leftarrow c_k, G_k \mid k = 1, \ldots, m\}$ be the set of all clauses in P_0 such that A' and A_j have the same predicate symbol. We consider two cases.

(R is empty) Since there is no clause of the form $A' \leftarrow G'$ in $ground(P_0 \cup Defs_n)$, there is no proof tree in $P_0 \cup Defs_n$ which uses γ_ν.
(R is non-empty) Let γ_ν be of the form $H \leftarrow A$ and let $A \leftarrow G$ be the clause in $ground(\{\gamma_h\})$ which is used at A in T, where $\gamma_h \in R$. Thus, there exists a clause γ'_h in P_j of the form $H' \leftarrow c, c_h, A' = A_h, G_h$. For all atoms B in G, we have that $stratum(B) \leq stratum(H)$ and there exists a proof tree T_B for B in $P_0 \cup Defs_n$ such that $size(T_B) < size(T)$. Thus, by hypothesis (IHsize) we have that there exists a proof tree T'_B for B in P_j. We construct T' as follows: we use $H \leftarrow G$ in $ground(\{\gamma_h\})$ at the root, and we use T'_B as a subtree at B, for all atoms B in G. For all negated atoms $\neg B$ in G we have that: (i) there is no proof tree for B in $P_0 \cup Defs_n$ because T is a proof tree, and (ii)$stratum(B) < stratum(H)$ because program $P_0 \cup Defs_n$ is locally stratified w.r.t. *stratum*. Thus, by hypothesis (IHstratum) we have that there is no proof tree for B in P_j. Thus, T' is a proof tree for H in P_j. □

Lemma 3.3.4. *If there exists a proof tree for H in P_j then there exists a weight-consistent proof tree for H in P_j.*

Proof. Let T be a proof tree for H in P_j of minimal size. Thus, for all atoms A in T, $size(T_A) = \mu(A)$, where T_A denotes the subtree of T rooted at A. Thus, for all atoms A and B in T such that B is a child of A we have $\mu(B) = size(T_B) < size(T_A) = \mu(A)$, that is, T is weight-consistent. □

Proposition 3.3.5. *Let P_0 be a locally stratified CLP program and let P_j,...,P_n be a transformation sequence which is obtained by applying the transformation rules of Section 3.2 with the conditions that the constrained atomic definition rule is not applied and the contextual constraint replacement rule R5n is only applied in its restricted form R5r. Then, for any ground atom H we have that: for all $i = j, \ldots, n$,*
(Soundness) if there exists a proof tree for H in P_i then there exists a proof tree for H in P_j, and
(Completeness) if there exists a proof tree for H in P_j then there exists a weight-consistent proof tree for H in P_i.

Proof. Let *stratum* be a stratification function for P_0 and for all the programs in the transformation sequence $P_0 \cup Defs_n, \ldots, P_j, \ldots, P_n$. The proof proceeds by induction on i.
Base case ($i = j$). Trivial.
Induction step. The inductive hypothesis is the following:
(*IHsoundness*) if there exists a proof tree for H in P_i then there exists a proof tree for H in P_j, and
(*IHcompleteness*1) if there exists a proof tree for H in P_j then there exists a weight-consistent proof tree for H in P_i.

By using Lemma 3.3.4 above, we have that (IHcompleteness1) is equivalent to:
(*IHcompleteness*) if there exists a weight-consistent proof tree for H in P_j then there exists a weight-consistent proof tree for H in P_i.
(*Soundness*) Given a proof tree T for H in P_{i+1} we construct a proof tree T' for H in P_i by well-founded induction on the lexicographic product of the well-founded order over *stratum* and the well-founded order over *size*.

Let $\gamma_\nu \in \mathit{ground}(\{\gamma\})$ be the ground clause of the form $H \leftarrow L_1, \ldots, L_k$ used at the root of T, where γ is a clause in P_{i+1} and ν is a valuation such that $\gamma_\nu = \nu(\gamma)$. The inductive hypotheses are:
(*IHstratum*) For all ground atoms A if $\mathit{stratum}(A) < \mathit{stratum}(H)$ then A has a proof tree in P_i iff A has a proof tree in P_{i+1}.
(*IHsize*) For all ground atoms A, if $\mathit{stratum}(A) \leq \mathit{stratum}(H)$ and A has a proof tree T_A in P_i such that $\mathit{size}(T_A) < \mathit{size}(T)$, then A has a proof tree T'_A in P_{i+1}.

We proceed by cases.

Case 1. Clause $\gamma \in P_i$. We begin the construction of T' by using γ_ν at the root. For all $h = 1, \ldots, k$ such that L_h is an atom A we have that $\mathit{size}(T_A) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_A for A in P_i which we use as subtree of T' at A. For all $h = 1, \ldots, k$ such that L_h is a negated atom $\neg B$ we have that $\mathit{stratum}(B) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for B in P_{i+1} because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for B in P_i. Thus, T' is a weight-consistent proof tree for H in P_i.

Case 2. Clause γ is obtained by positive unfolding (rule R2p). Thus, there exist (renamed apart) clauses α and β in P_i of the form $H \leftarrow c, G_L, A, G_R$ and $B \leftarrow d, G$, respectively, such that clause γ is of the form $H \leftarrow c, d, A = B, G_L, G, G_R$. Since $\gamma_\nu \in \mathit{ground}(\{\gamma\})$ we have that $\mathcal{D} \models \nu(c \wedge d \wedge A = B)$ and thus, $\alpha_\nu \in \mathit{ground}(P_i)$ and $\beta_\nu \in \mathit{ground}(P_i)$.
We construct T' as follows: we use α_ν at the root, then we use β_ν at $\nu(A)$ (which is equal to $\nu(B)$). The leaves of the current proof tree are $L_1, \ldots, L_k$. For all $h = 1, \ldots, k$ such that L_h is an atom C and T_C is the subtree of T rooted at C, we have that $\mathit{stratum}(C) \leq \mathit{stratum}(H)$ and $\mathit{size}(T_C) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_C for C in P_i which we use as a subtree of T' at C. For all $h = 1, \ldots, k$ such that L_h is a negated atom $\neg C$, we have that $\mathit{stratum}(C) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_{i+1} because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_i. Thus, T' is a proof tree for H in P_i.

Case 3. Clause γ is obtained by negative unfolding (rule R2n - Case F). Let clause γ be of the form $H' \leftarrow c, L'_1, \ldots, L'_k$. Thus, there exists a clause α in P_i of

the form $H' \leftarrow c, L'_1, \ldots, L'_f, \neg C', L'_{f+1}, \ldots, L'_k$ such that there is no clause in P_i defining the predicate which occurs in C'. We begin the construction of T' by using a ground instance of α_ν of the form $H \leftarrow L_1, \ldots, L_f, \neg C, L_{f+1}, \ldots, L_k$ at the root. For all $h = 1, \ldots, k$ such that L_h is an atom A and T_A is the subtree of T rooted at A, we have that $\mathit{stratum}(A) \leq \mathit{stratum}(H)$ and $\mathit{size}(T_A) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_A for A in P_i which we use as a subtree of T' at A. For all $h = 1, \ldots, k$ such that L_h is a negated atom $\neg B$, we have that $\mathit{stratum}(B) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for B in P_{i+1} because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for B in P_i. Moreover, since no clause in P_i defines the predicate of C, we have that there is no proof tree for C in P_i and $\mathit{stratum}(C) < \mathit{stratum}(H)$ because program P_{i+1} is locally stratified w.r.t. *stratum*. By hypothesis (IHstratum) there is no proof tree for C in P_i, and thus, T' is a proof tree for H in P_i.

Case 4. Clause γ is obtained by folding (rule R3n). Let γ be obtained by folding a clause α in P_i of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, L', L'_{f+1}, \ldots, L'_k$ by using a (variant of a) definition clause $\delta \in \mathit{Defs}_n$ of the form $A' \leftarrow d, B'$.
(Case P) Literal L' is the atom B' and clause γ is of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, A', L'_{f+1}, \ldots, L'_k$.

Let γ_ν be of the form $H \leftarrow L_1, \ldots, L_{f-1}, A, L_{f+1}, \ldots, L_k$. By hypothesis (IHsize) there exists a proof tree for A in P_i. Thus, by hypothesis (IHsoundness) there exists a proof tree for A in P_j. By Proposition 3.3.3, there exists a proof tree for A in $P_0 \cup \mathit{Defs}_n$ which uses a clause $A \leftarrow B$ in $\mathit{ground}(\{\delta\})$ at the root. Thus, by Proposition 3.3.3, there exists a proof tree for B in P_j. By hypothesis (IHcompleteness) there exists a proof tree T'_B for B in P_i. By the conditions on the applicability of rule R3n, there exist a clause α_σ in $\mathit{ground}(\{\alpha\})$ of the form $H \leftarrow L_1, \ldots, L_{f-1}, B, L_{f+1}, \ldots, L_k$ and a valuation σ such that $\alpha_\sigma = \sigma(\nu(\alpha))$. We begin the construction of T' by using α_σ at the root. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is an atom C and T_C is the subtree of T rooted at C, we have that $\mathit{stratum}(C) \leq \mathit{stratum}(H)$ and $\mathit{size}(T_C) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_C for C in P_i which we use as a subtree of T' at C. We complete the construction of T' by using T'_B as a subtree at B. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is a negated atom $\neg C$ we have that $\mathit{stratum}(C) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_{i+1} because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_i. Thus, T' is a proof tree for H in P_i.
(Case N) Literal L' is the negated atom $\neg B'$ and clause γ is of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, \neg A', L'_{f+1}, \ldots, L'_k$.

Let γ_ν be of the form $H \leftarrow L_1, \ldots, L_{f-1}, \neg A, L_{f+1}, \ldots, L_k$. Since program P_{i+1} is locally stratified w.r.t. *stratum*, we have that $\mathit{stratum}(A) < \mathit{stratum}(H)$

and there is no proof tree for A in P_{i+1} because T is a proof tree. Thus, by hypothesis (IHstratum) there exists no proof tree for A in P_i. By hypothesis (IHcompleteness) there exists no proof tree for A in P_j. By Proposition 3.3.3, there exists no proof tree for A in $P_0 \cup \mathit{Defs}_n$. Thus, for all clauses $A \leftarrow D$ in $\mathit{ground}(\{\delta\})$ we have that there is no proof tree for D in $P_0 \cup \mathit{Defs}_n$. By Proposition 3.3.3, for all clauses $A \leftarrow D$ in $\mathit{ground}(\{\delta\})$ there exists no proof tree for D in P_j and, by hypothesis (IHsoundness), there exists no proof tree for D in P_i. By the conditions on the applicability of rule R3n, there exist a clause α_σ in $\mathit{ground}(\{\alpha\})$ of the form $H \leftarrow L_1, \ldots, L_{f-1}, \neg B, L_{f+1}, \ldots, L_k$ and a valuation σ such that $\alpha_\sigma = \sigma(\nu(\alpha))$. We begin the construction of T' by using α_σ at the root. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is an atom C and T_C is the subtree of T rooted at C, we have that $\mathit{stratum}(C) \leq \mathit{stratum}(H)$ and $\mathit{size}(T_C) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_C for C in P_i which we use as a subtree of T' at C. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is a negated atom $\neg C$ we have that $\mathit{stratum}(C) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_{i+1}, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_i. Moreover, by the conditions on the applicability of rule R3n, for every clause in $\mathit{ground}(\{\alpha\})$ of the form $H \leftarrow L_1, \ldots, L_{f-1}, \neg B, L_{f+1}, \ldots, L_k$ we have that $A \leftarrow B$ is in $\mathit{ground}(\{\delta\})$, and thus, there exists no proof tree for B in P_i. Thus, T' is a proof tree for H in P_i.

Case 5. Clause γ is obtained by constraint replacement (rule R5r). Let η be the clause in P_i which has been replaced by γ. By the condition on the applicability of rule R5r we have that $\mathit{ground}(\{\gamma\}) = \mathit{ground}(\{\eta\})$. We begin the construction of T' by using γ_ν at the root. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is an atom C and T_C is the subtree of T rooted at C, we have that $\mathit{stratum}(C) \leq \mathit{stratum}(H)$ and $\mathit{size}(T_C) < \mathit{size}(T)$. By hypothesis (IHsize) there exists a proof tree T'_C for C in P_i which we use as a subtree of T' at C. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is a negated atom $\neg C$, we have that $\mathit{stratum}(C) < \mathit{stratum}(H)$, because program P_{i+1} is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_{i+1}, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_i. Thus, T' is a proof tree for H in P_i.

(*Completeness*) Given a weight-consistent proof tree T for H in P_i we construct a weight-consistent proof tree T' for H in P_{i+1} by well-founded induction on the lexicographic product of the well-founded order over *stratum* and the well-founded order over *size*.

Let $\gamma_\nu \in \mathit{ground}(\{\gamma\})$ be the ground clause of the form $H \leftarrow L_1, \ldots, L_k$ used at the root of T, where γ is a clause in P_i and ν is a valuation such that $\gamma_\nu = \nu(\gamma)$. The inductive hypotheses are:

(*IHstratum*) For all ground atoms A if $stratum(A) < stratum(H)$ then A has a proof tree in P_i iff A has a proof tree in P_{i+1}.
(*IHweight*) For all ground atoms A such that $stratum(A) \leq stratum(H)$ and $\mu(A) < \mu(H)$, if A has a weight-consistent proof tree T_A in P_i then A has a weight-consistent proof tree T'_A in P_{i+1}.

We proceed by cases.

Case 1. Clause $\gamma \in P_{i+1}$. We begin the construction of T' by using γ_ν at the root. For all $h = 1, \ldots, k$ such that L_h is an atom A, we have that $stratum(A) \leq stratum(H)$, because P_i is locally stratified w.r.t. *stratum*, and $\mu(A) < \mu(H)$, because T is weight-consistent. By hypothesis (IHweight) there exists a weight-consistent proof tree T'_A for A in P_{i+1} which we use as subtree of T' at A. For all $h = 1, \ldots, k$ such that L_h is a negated atom $\neg B$, we have that $stratum(B) < stratum(H)$, because program P_i is locally stratified w.r.t. *stratum*, and there is no proof tree for B in P_i, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for B in P_{i+1}. Thus, T' is a weight-consistent proof tree for H in P_{i+1}.

Case 2. Clause γ is removed by positive unfolding (rule R2p). Let γ be a clause of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, p(X), L'_{f+1}, \ldots, L'_k$ such that there is no clause in P_i whose head has the predicate symbol p. Since there is no clause of the form $p(t) \leftarrow G$ in $ground(P_i)$, there is no proof tree in P_i which uses γ_ν.

Case 3. Clause γ is removed by negative unfolding (rule R2n - Case V). Let clause γ be of the form $H' \leftarrow c, G_L, \neg p(Y), G_R$. By the conditions on Case V of the negative unfolding rule, there exists a clause $p(X) \leftarrow$ in P_i. Thus, every ground instance of $p(X)$ has a proof tree in P_i. Thus, there is no proof tree in P_i which uses γ_ν.

Case 4. Clause γ is removed by folding (rule R3n). Let γ be a clause of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, L', L'_{f+1}, \ldots, L'_k$ and let α be the clause in P_i which is obtained by folding γ w.r.t. L', by using a (variant of a) definition clause $\delta \in Defs_n$ of the form $A' \leftarrow d, B'$.
(Case P) Literal L' is the atom B' and clause α is of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, A', L'_{f+1}, \ldots, L'_k$.

Let γ_ν be of the form $H \leftarrow L_1, \ldots, L_{f-1}, B, L_{f+1}, \ldots, L_k$. By hypothesis (IHsoundness), there exists a proof tree for B in P_j. Let T''_B be a proof tree for B in P_j of minimal size. Let A be an atom such that $A \leftarrow B$ is a clause in $ground(\{\delta\})$. By construction of P_j, we have that the tree which is obtained from T''_B by replacing the root B by A is a proof tree for A in P_j. Thus, we have that $\mu(A) \leq \mu(B)$ and, by Lemma 3.3.4, there exists a weight-consistent proof tree for A in P_j. By hypothesis (IHcompleteness), there exists a weight-consistent proof tree for A in P_i. Since T is weight-consistent, we have $\mu(A) \leq \mu(B) < \mu(H)$. Moreover, $stratum(A) \leq stratum(H)$, because P_{i+1} is locally stratified w.r.t. *stratum*. By hypothesis (IHweight), there exists

a weight-consistent proof tree T'_A for A in P_{i+1}. By the conditions on the applicability of rule R3n, there exist a clause α_σ in $ground(\{\alpha\})$ of the form $H \leftarrow L_1, \ldots, L_{f-1}, A, L_{f+1}, \ldots, L_k$ and a valuation σ such that $\alpha_\sigma = \sigma(\nu(\alpha))$. We use α_σ at the root of T'. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is an atom C, we have that $stratum(C) \leq stratum(H)$, because P_i is locally stratified w.r.t. *stratum*, and $\mu(C) < \mu(H)$, because T is weight-consistent. By hypothesis (IHweight), there exists a weight-consistent proof tree T'_C for C in P_{i+1} which we use as subtree of T' at C. We complete the construction of T' by using T'_A as subtree at A. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is a negated atom $\neg C$, we have that $stratum(C) < stratum(H)$, because program P_i is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_i, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_{i+1}. Thus, T' is a weight-consistent proof tree for H in P_{i+1}.
(Case N) Literal L' is the negated atom $\neg B'$ and clause α is of the form $H' \leftarrow c, L'_1, \ldots, L'_{f-1}, \neg A', L'_{f+1}, \ldots, L'_k$.

Let γ_ν be of the form $H \leftarrow L_1, \ldots, L_{f-1}, \neg B, L_{f+1}, \ldots, L_k$. Since T is a proof tree, there is no proof tree for B in P_i. Thus, by hypothesis (IHcompleteness), there exists no proof tree for B in P_j and, by Proposition 3.3.3, there exists no proof tree for B in $P_0 \cup Defs_n$. By the conditions on the applicability of rule R3n, there exist a valuation σ such that $\alpha_\sigma = \sigma(\nu(\alpha))$ is a clause of the form $H \leftarrow L_1, \ldots, L_{f-1}, \neg A, L_{f+1}, \ldots, L_k$ and a clause $A \leftarrow B$ in $ground(\{\delta\})$ such that there exists no proof tree for A in $P_0 \cup Defs_n$. By Proposition 3.3.3, there exists no proof tree for A in P_j and, by hypothesis (IHsoundness), there exists no proof tree for A in P_i. Since program P_{i+1} is locally stratified w.r.t. *stratum*, we have that $stratum(A) < stratum(H)$. Thus, by hypothesis (IHstratum) there exists no proof tree for A in P_{i+1}. We begin the construction of T' by using α_σ at the root. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is an atom C we have that $stratum(C) \leq stratum(H)$, because P_i is locally stratified w.r.t. *stratum*, and $\mu(C) < \mu(H)$, because T is weight-consistent. By hypothesis (IHweight) there exists a weight-consistent proof tree T'_C for C in P_i which we use as subtree of T' at C. For all $h = 1, \ldots, f-1, f+1, \ldots, k$ such that L_h is a negated atom $\neg C$, we have that $stratum(C) < stratum(H)$, because program P_i is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_i, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_{i+1}. Thus, T' is a weight-consistent proof tree for H in P_{i+1}.

Case 5. Clause γ is removed by removal of clause with unsatisfiable body (rule R4f). By the condition on rule R4f we have that $ground(\{\gamma\}) = \emptyset$. Thus, there is no γ_ν.

Case 6. Clause γ is removed by removal of subsumed clause (rule R4s). Let γ

be of the form $H \leftarrow c, G$. By the condition on rule R4s there exists a clause λ of the form $p(X) \leftarrow$ in P_{i+1} where p is the predicate symbol of H. Thus, there exists $H \leftarrow$ in $ground(\{\lambda\})$ which we use to construct the weight-consistent proof tree T' for H in P_{i+1}.

Case 7. Clause γ is removed by removal of useless clauses (rule R4u). If p is a useless predicate of P_i and H is an atom with predicate p, then there is no proof tree for H in P_i. The proof proceeds by contradiction. We assume that there is a proof tree T for H in P_i. By the definition of useless predicate, each node in T has a son which is a positive literal of the form $q(\ldots)$, where q is a useless predicate of P_i. Thus, there exists a leaf of the form $q(\ldots)$, and this contrasts with the hypothesis that T is a proof tree.

Case 8. Clause γ is removed by constraint replacement (rule R5r). Let η be the clause in P_{i+1} which replaces γ. By the condition on rule R5r we have that $ground(\{\gamma\}) = ground(\{\eta\})$. We begin the construction of T' by using γ_ν at the root. For all $h = 1, \ldots, k$ such that L_h is an atom C, we have that $stratum(C) \leq stratum(H)$, because P_i is locally stratified w.r.t. *stratum*, and $\mu(C) < \mu(H)$, because T is weight-consistent. By hypothesis (IHweight) there exists a weight-consistent proof tree T'_C for C in P_i which we use as subtree of T' at C. For all $h = 1, \ldots, k$ such that L_h is a negated atom $\neg C$ we have that $stratum(C) < stratum(H)$, because program P_i is locally stratified w.r.t. *stratum*, and there is no proof tree for C in P_i, because T is a proof tree. By hypothesis (IHstratum) we have that there is no proof tree for C in P_{i+1}. Thus, T' is a weight-consistent proof tree for H in P_{i+1}. □

Definition 3.3.6. [*Depth of a Proof Tree*] The *depth* of a proof tree T is $depth(T) = \max\{length(\pi) \mid \pi$ is a path from the root of T to a leaf of $T\}$.

Lemma 3.3.7. *Let P be a definite constraint logic program. For all ground atoms A and for all $k \geq 1$, we have*

A has a proof tree T in P such that $depth(T) \leq k$ iff $A \in T_P \uparrow k$.

Proof. By induction on k.
Base case ($k = 1$). There exists a proof tree T for A in P such that $depth(T) = 1$ iff there exists a clause of the form $A \leftarrow$ in $ground(P)$ iff $A \in T_P \uparrow 1$.
Induction step.

(If part) Let γ be a clause of the form $A \leftarrow A_1, \ldots, A_n$ in $ground(P)$ such that $A_i \in T_P \uparrow (k-1)$, for all $i = 1, \ldots, n$. Thus, by inductive hypothesis, there exists a proof tree T_i for A_i in P such that $depth(T_i) \leq k-1$, for all $i = 1, \ldots, n$. We construct T by using γ at the root and by using T_i as a subtree of T at A_i. Thus, T is a proof tree for A in P and $depth(T) \leq k$.

(Only if part) Let T be a proof tree for A in P such that $depth(T) = k$, and let $A \leftarrow A_1, \ldots, A_n$ be the clause in $ground(P)$ used at the root of T. For all

$i = 1, \ldots, n$, let T_i be the subtree of T rooted at A_i. Since A_i is an atom, T_i is a proof tree for A_i in P and $depth(T_i) \leq k - 1$, we can apply the inductive hypothesis and we have that $A_i \in T_P \uparrow (k-1)$, for all $i = 1, \ldots, n$. Thus, $A \in T_P \uparrow k$. □

Corollary 3.3.8. [Proof Trees and Least $\mathcal{D}$-model] *Let P be a definite constraint logic program. For all ground atoms A, we have*

A has a proof tree in P iff $A \in lm(P, \mathcal{D})$

Proof. For all atoms A, we have $A \in lm(P, \mathcal{D})$ iff $A \in T_P \uparrow \omega$ iff there exists $k \geq 1$ such that $A \in T_P \uparrow k$. By using Lemma 3.3.7, $A \in lm(P, \mathcal{D})$ iff there exists $k \geq 1$ such that A has a proof tree T in P and $depth(T) \leq k$ iff A has a proof tree in P. □

Theorem 3.3.9. [Proof Trees and Perfect Model] *Let P be a constraint logic program which is locally stratified w.r.t.* stratum. *For all ground atoms A, we have*

A has a proof tree in P iff $A \in M(P)$

Proof. Let $stratum(A) = \alpha$ and let T be a proof tree for A in P. Thus, by definition of perfect model, there exists a clause $\gamma \in ground(P)$ of the form $A \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$, with $m, n \geq 0$, which is used in T iff there exists a ground clause $\gamma' \in P'_\alpha$ of the form $A \leftarrow A_1, \ldots, A_n$. The thesis follows from Corollary 3.3.8. □

Theorem 3.3.10. [Correctness of the Transformation Rules] *Let $P_0, \ldots, P_n$ be a transformation sequence. Let us assume that*
(i) P_0 is locally stratified, and
during the construction of $P_0, \ldots, P_n$
(ii) 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,
(iii) the contextual constraint replacement rule R5n is only applied in its restricted form R5r,
Then, there exists a function stratum *such that programs P_0 and P_n are locally stratified w.r.t.* stratum *and*

$M(P_0 \cup Defs_n) = M(P_n)$

where $Defs_n$ denotes the set of definitions introduced during the construction of $P_0, \ldots, P_n$.

Proof. It follows from Theorem 3.3.2, Proposition 3.3.3, Proposition 3.3.5 and Theorem 3.3.9. □

In order to state the theorem on the correctness of the contextual constraint replacement rule R5n w.r.t. the perfect model we need to extend the definition of *call patterns* of a clause presented in Section 2.3 to general constraint logic programs.

Definition 3.3.11. [*General Call Patterns*] Given a clause γ of the form $p(X) \leftarrow d, L_1, \ldots, L_k$, with $k > 0$, the set of *general call patterns* of γ, which is denoted by $CP(\gamma)$, is the set of triples $\langle solve(d, Y), A, Y \rangle$ such that: either (i) L_j is the atom A, for some $j = 1, \ldots, k$, and Y denotes the linking variables of A in γ, or (ii) L_j is the negated atom $\neg A$, for some $j = 1, \ldots, k$, and $Y = vars(A)$. $\langle solve(d, Y), A, Y \rangle$ is said to be the call pattern of γ for L_j.

General call patterns will be used in our contextual specialization strategy below (see Section 3.4) for introducing new definitions and for applying the contextual constraint replacement rule R5n.

Lemma 3.3.12. [CCR] *Let $P_0, \ldots, P_n$ be a transformation sequence such that, for all $i = 0, \ldots, n-1$, program P_{i+1} is derived from P_i by applying the contextual constraint replacement rule R5n w.r.t. a given set $\mathbf{C}$ of constrained atoms such that $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X \rangle \in CP(P_0)\}$.*
Then, for all constrained atoms $c, A \in \mathbf{C}$, for every valuation ν and for every clause γ such that $hd(\gamma) = \nu(A)$, we have that if $\mathcal{D} \models \nu(c)$ then, for all i,
$\gamma \in ground(P_0)$ iff $\gamma \in ground(P_i)$.

Proof. By induction on i.
Base case ($i = 0$). Trivial.
Induction step. We proceed by cases.

Case 1. γ is a ground instance of a clause which is in P_i and in P_{i+1}.
Trivial.

Case 2. γ is a ground instance of a clause which is removed from P_i by contextual constraint replacement.

Let $\gamma \in ground(\{\alpha\})$ where α is a variant of a clause in P_i of the form $A \leftarrow d, G$ which is replaced by a clause β in P_{i+1} of the form $A \leftarrow d', G$. From the condition on rule R5n we have that $\mathcal{D} \models \forall X(c \rightarrow (\exists W\, d \leftrightarrow \exists W'\, d')$ where $Y = FV(A, G)$, $W = FV(d)$–Y and $W' = FV(d')$–Y. Thus, $\gamma \in ground(\{\alpha\})$ iff $\gamma \in ground(\{\beta\})$. The thesis follows by inductive hypothesis. □

Proposition 3.3.13. [CCR] *Let $P_0, \ldots, P_n$ be a transformation sequence such that, for all $i = 0, \ldots, n-1$, program P_{i+1} is derived from P_i by applying the contextual constraint replacement rule R5n w.r.t. a given set $\mathbf{C}$ of constrained atoms such that $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X \rangle \in CP(P_0)\}$.*
Then, for all constrained atoms $c, A \in \mathbf{C}$, for every valuation ν and for every clause γ such that $hd(\gamma) = \nu(A)$, we have that if $\mathcal{D} \models \nu(c)$ then, for all i,
T is a proof tree for $\nu(A)$ in P_0 iff T is a proof tree for $\nu(A)$ in P_i

Proof. By induction on i.
Base case ($i = 0$). Trivial.
Induction step.

(*Soundness*) We proceed by well-founded induction on $size(T)$. The inductive hypothesis is:
(*IHsize*) for all constrained atoms $d, B \in \mathbf{C}$ and for every valuation θ we have that if $\mathcal{D} \models \theta(d)$, T' is a proof tree for $\theta(B)$ in P_{i+1} and $size(T') < size(T)$ then T' is a proof tree for $\theta(B)$ in P_i.

Let γ be the clause used at the root of T. By Lemma 3.3.12 above, $\gamma \in ground(P_0)$. Thus, for all atoms B' in $bd(\gamma)$ we have that there exist a call pattern d, B in $\mathbf{C}$ and a valuation θ such that $\mathcal{D} \models \theta(d)$ and $B' = \theta(B)$, because $\mathbf{C}$ contains all the call patterns of P_0. Let T' be the subtree of T rooted at B'. By hypothesis (IHsize), we have that T' is a proof tree for B' in P_i. Moreover, by Lemma 3.3.12, we have that $\gamma \in ground(P_i)$. Thus, T is a proof tree for $\nu(A)$ in P_i.

(*Completeness*) We proceed by well-founded induction on $size(T)$. The inductive hypothesis is:
(*IHsize*) for all constrained atoms $d, B \in \mathbf{C}$ and for every valuation θ we have that if $\mathcal{D} \models \theta(d)$, T' is a proof tree for $\theta(B)$ in P_i and $size(T') < size(T)$ then T' is a proof tree for $\theta(B)$ in P_{i+1}.

Let γ be the clause used at the root of T. By Lemma 3.3.12 above, $\gamma \in ground(P_0)$. Thus, for all atoms B' in $bd(\gamma)$ we have that there exist a call pattern d, B in $\mathbf{C}$ and a valuation θ such that $\mathcal{D} \models \theta(d)$ and $B' = \theta(B)$, because $\mathbf{C}$ contains all the call patterns of P_0. Let T' be the subtree of T rooted at B'. By hypothesis (IHsize), we have that T' is a proof tree for B' in P_{i+1}. Moreover, by Lemma 3.3.12, we have that $\gamma \in ground(P_{i+1})$. Thus, T is a proof tree for $\nu(A)$ in P_{i+1}. □

Theorem 3.3.14. [Correctness of the Contextual Constraint Replacement Rule R5n] *Let $P_0, \ldots, P_n$ be a transformation sequence such that, for all $i = 0, \ldots, n-1$, program P_{i+1} is derived from P_i by applying the contextual constraint replacement rule R5n w.r.t. a given set $\mathbf{C}$ of constrained atoms such that $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X\rangle \in CP(P_0)\}$. Assume that programs P_0 and P_n are locally stratified w.r.t.* stratum.
Then, for all constrained atoms $c, A \in \mathbf{C}$ and for every valuation ν we have that:

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

Proof. It follows from Proposition 3.3.13 and Theorem 3.3.9. □

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 3.3.14 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, all applications of the contextual constraint replacement rule presented in this thesis, do preserve local stratification. This is due to the fact that, according to the specialization strategies presented in Sections 2.5.3, 3.4 and 4.4, 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 3.3.14.

Corollary 3.3.15. [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 $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X\rangle \in CP(P_0)\}$. Then* (i) *$P_n$ is stratified and* (ii) *for all constrained atoms $c, A \in \mathbf{C}$ and for every valuation ν, we have that:*

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

3.4 An Automated Strategy for Contextual Specialization of General Constraint Logic Programs

We now describe a parameterized strategy for specializing CLP($\mathcal{D}$) programs with locally stratified negation. This strategy is a proper extension of the strategy for the specialization of definite constraint logic programs presented in Section 2.5.

As in the definite case, the strategy for specializing general constraint logic programs is divided into three phases and is parameterized by: (i) the function *solve* which is used for the application of the constraint replacement rule, (ii)

an *unfolding function Unfold* for guiding the unfolding process, (iii) a well-quasi order $\preceq_u$ over constrained goals which tells us when to terminate the unfolding process, (iv) a clause generalization function *gen*, with its associated widening operator ∇, and (v) a well-quasi order $\preceq_g$ over constrained atoms which tells us when to activate the clause generalization process. Once the choice of these parameters has been made, our strategy can be applied in a fully automatic way.

We introduce a tree *Defstree*, called *definition tree,* whose nodes are the clauses introduced by the definition rule during program specialization. Moreover, for each clause δ in *Defstree* we introduce a tree *Utree*(δ), called *unfolding tree*. The root of *Utree*(δ) is δ itself, and the nodes of *Utree*(δ) are the clauses derived from δ by applying the positive unfolding and constraint replacement rules.

The strategy for specializing general constraint logic programs can be presented in a way which is very similar to the presentation of the corresponding strategy for the definite case. We now discuss on the similarities and the differences between the two strategies, in each phase.

Phase A. We consider a general CLP program P and a constrained atom c, A and we iterate the procedures *Unfold-Replace* and *Define-Fold* as we now explain.

During the *Unfold-Replace* procedure we apply the *positive* unfolding rule according to a given unfolding function, and we solve the constraints in the derived clauses by using the given function *solve*. During this phase, we never apply the negative unfolding rule.

This procedure is very similar to the *Unfold-Replace* procedure presented in the strategy for specializing definite CLP programs. In fact, we can even reuse the same pseudo-code of Section 2.5.1 for (i) the *Unfold-Replace* procedure and (ii) the *Unfold* and *Replace* functions by modifying the definition of clause with non-failing body as follows. A clause of the form $H \leftarrow c, L_1, \ldots, L_n$ has a *non-failing* body iff c is satisfiable, and for $i = 1, \ldots, n$, if L_i is A then A is not failed, otherwise, if L_i is $\neg A$ then A is not valid. Notice that the unfolding function is not defined on clauses of the form $H \leftarrow c, G$ where G is a conjunction of negated atoms. The termination of this procedure is ensured by the use of the well-quasi order $\preceq_u$.

We then apply the *Define-Fold* procedure and we fold the clauses we have derived during the *Unfold-Replace* procedure. For folding we make use of already available definitions and, possibly, some new definitions introduced by using the clause generalization function.

The definition to be used for folding is selected according to the definition function *Define* of Section 2.5.2 where we replace the notion of call pattern by the notion of general call pattern presented in Section 3.3. This is possible

because the general call pattern of a clause for a negated atom $\neg A$ is a triple of the form $\langle c, A, Y\rangle$, where A is an atom. Notice that (i) the clauses introduced by the definition function are definite clauses, and (ii) when applying the constrained atomic folding rule of Section 3.2 w.r.t. a negated atom $\neg A$ (see Case (N)), we introduce a new clause where $\neg A$ has been replaced by a negated atom, and not by an atom, as illustrated by the following simple example.

Example. Let γ be a clause of the form

$q(X, Y) \leftarrow X \geq 1, Y > X, \underline{\neg p(Y)}$

and let δ be a definition of the form

$newp(Y) \leftarrow Y \geq 0, p(Y)$

then, by constrained atomic folding γ w.r.t. $\neg p(Y)$ using δ we derive a clause of the form

$q(X, Y) \leftarrow X \geq 1, Y > X, \underline{\neg newp(Y)}$

Case (N) of our constrained atomic folding rule is not an instance of the folding rule presented in [74] for general logic programs, where negated atoms are replaced by atoms. Indeed, an instance of that rule would introduce a definition δ' of the form

$newp2(Y) \leftarrow Y \geq 0, \neg p(Y)$

for folding clause γ above w.r.t. $\neg p(Y)$, thereby deriving a clause of the form

$q(X, Y) \leftarrow X \geq 1, Y > X, \underline{newp2(Y)}$

Moreover, in order to unfold the literal $\neg p(Y)$ in δ' we would need to introduce a negative unfolding rule with a much weaker applicability condition. Recall that a negative unfolding rule was presented in Section 3.2, but it can only be applied w.r.t. negated atoms of the form $\neg A$, where A is a valid atom.

Thus, for *Define-Fold* procedure, we use the same pseudo-code of Section 2.5.2, except for some minor differences which are highlighted below. (1) The phrase 'Let λ be of the form $A_0 \leftarrow d, A_1, \ldots, A_k$' should be replaced by the phrase 'Let λ be of the form $A_0 \leftarrow d, L_1, \ldots, L_k$' because, in general CLP programs the body of a clause contains literals, not only atoms. (2) The phrase 'Let cp_i be the call pattern of γ_i for A_i' should be replaced by the phrase 'Let cp_i be the general call pattern of γ_i for L_i'. (3) The phrase 'Fold γ_i w.r.t. A_i' should be replaced by the phrase 'Fold γ_i w.r.t. L_i'.

Phase A terminates with output program P_A when no new definitions need to be introduced for performing the folding steps. The termination of Phase A is ensured by the properties of the generalization function and well-quasi order $\preceq_g$ which guarantee that the set of generated definitions is finite.

Phase B. We consider program P_A and, by applying the contextual constraint replacement rule, from each clause defining a predicate, say q, we remove the constraints which hold before the execution of the clause. These constraints can be determined by computing the least upper bound of the set of constraints

which occur in the clauses containing a call of q. The presentation of this phase is almost identical to that of Section 2.5.3 except for the fact that the notion of call pattern has been replaced by the notion of *general* call pattern.

Phase C. If the output of Phase B is a program P_B which admits a finite stratification $\{S_1, \ldots, S_n\}$, then, during Phase C, we apply the following rules: (i) positive and negative unfolding, (ii) removal of useless and subsumed clauses, and (iii) constraint replacement. This phase differs from Phase C of Section 2.5.3 in that we apply also the negative unfolding rule and we iterate over the strata $S_1, \ldots, S_n$ of program P_B.

We now present our strategy for contextual specialization of CLP($\mathcal{D}$) programs.

Contextual Specialization Strategy

Input: (i) A CLP($\mathcal{D}$) program P and
(ii) a constrained atom $c, p(X)$ such that $FV(c) \subseteq X$.
Output: A CLP($\mathcal{D}$) program P_s and an atom $p_s(X)$.

Phase A. By the definition rule introduce a clause δ_0 of the form $p_s(X) \leftarrow c, p(X)$. Let *Defstree* consist of clause δ_0 only.

$P_s := \emptyset$; $\quad NewDefs := \{\delta_0\}$;

while $NewDefs \neq \emptyset$ **do**
  Unfold-Replace($NewDefs$, $UForest$);
  Define-Fold($UForest$, $Defstree$, $NewDefs$, $FoldedCls$);
  $P_s := P_s \cup FoldedCls$
end-while

Phase B. [*Contextual Constraint Replacement*]
Let P_s be a program of the form $\{\gamma_1, \ldots, \gamma_p\}$ and
let $\mathbf{C}$ be the set $\{(solve(c, X), p_s(X))\} \cup \{(d, A) \mid \langle d, A, Y\rangle \in CP(P_s)\}$ of constrained atoms.
for $i = 1, \ldots, p$ **do**
  Let γ_i be a clause of the form $q(X) \leftarrow e_1, \ldots, e_n, G$
  where $e_1, \ldots, e_n$ are basic constraints with free variables in $X \cup vars(G)$;
  Let $\mathbf{C}_q$ be the set $\{(d_1, q(X)), \ldots, (d_k, q(X))\}$ of all renamed
  constrained atoms $d, q(X)$ in $\mathbf{C}$;
  Let f be the conjunction of all e_j's such that
  $\mathcal{D} \models \forall\, (d_r \rightarrow e_j)$ does not hold;
  Apply the contextual constraint replacement rule w.r.t. C_q
  thereby replacing γ_i by the clause $q(X) \leftarrow f, G$;
endfor

Phase C. This phase is performed only if the program P_B admits a finite stratification $\{S_1, \ldots, S_n\}$. In this case, by working bottom-up on the strata

$S_1, \ldots, S_n$, we simplify the definition of every predicate p in P_B, with the aim of deriving either the fact $p(\ldots) \leftarrow$ or the empty definition. During this phase we apply the following rules: (i) positive and negative unfolding, (ii) removal of useless and subsumed clauses, and (iii) constraint replacement.
The algorithm for Phase C is as follows.
$P_s := \emptyset$
for $i := 1, \ldots, n$ **do**
  repeat
    $S' := S_i$;
    Apply to S_i, as long as possible, the rule for removing subsumed clauses;
    Apply to S_i, as long as possible, the negative unfolding rule and
    the positive unfolding rule w.r.t. valid and failed atoms 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 $S' = S_i$
  Remove the useless clauses from S_i;
  $P_s := P_s \cup S_i$;
end-for □

3.5 Correctness of the Strategy

Theorem 3.5.1. [Correctness of the Contextual Specialization Strategy] *Let P be a locally stratified general CLP($\mathcal{D}$) program and $c, p(X)$ be a constrained atom with $FV(c) \subseteq X$. Let P_s and $p_s(X)$ be the general CLP($\mathcal{D}$) program and the atom obtained by the contextual specialization strategy. Let the output of Phase B be a program P_B.*
Then, if there exists a function stratum *such that P and P_B are locally stratified w.r.t.* stratum*, then program P_s is locally stratified w.r.t.* stratum *and for every valuation ν we have that:*

$$\text{if } \mathcal{D} \models \nu(c) \text{ then } \quad \nu(p(X)) \in M(P) \text{ iff } \nu(p_s(X)) \in M(P_s)$$

Proof. During the application of the contextual specialization strategy, folding is applied only to clauses which have been derived by one or more applications of the unfolding rule, followed by applications of the constraint replacement rule. Thus, the thesis follows from Theorem 3.3.2, Theorem 3.3.10 and Theorem 3.3.14 (see Section 3.3). □

3.6 Termination of the Strategy

Theorem 3.6.1. [Termination of the Contextual Specialization Strategy] *Let P be a general CLP($\mathcal{D}$) program, and $c, p(X)$ be a constrained atom with $FV(c) \subseteq X$. If the widening operator ∇ used for clause generalization agrees with the well-quasi order $\preceq_g$, then the contextual specialization strategy terminates.*

Proof. (Outline) It is similar to the proof of termination of the contextual specialization strategy for definite constraint logic programs (see Theorem 2.7.2). □

3.7 Related Work

Tamaki and Sato's unfold/fold transformation rules [80] have been extended to logic programs with negation by Seki [74]. Seki's rules have been proved to preserve the perfect model of stratified programs [74] and the well-founded model of general programs [73, 75]. A set of unfold/fold transformation rules which preserve the perfect model of constraint logic programs with stratified negation has been proposed by Maher in [51]. Other work (see, for instance, [7, 33, 62, 68]) presents variants of the transformation rules, which preserve various semantics of negation, including the semantics based on the Clark Completion, the operational semantics based on SLDNF resolution, the perfect model semantics, the stable model semantics, and the well-founded model semantics.

The rules considered in this chapter are adaptations, tailored to the task of program specialization, of the transformation rules presented in the literature. However, some of our rules are not simply instances or combinations of already known transformation rules.

Let us examine our rules in more detail. The positive unfolding rule (R2p), the rule for removal of clauses with unsatisfiable body (R4f), the rule for removal of subsumed clauses (R4s), and the constraint replacement rule (R5r), are identical to rules proposed by Maher [51]. The negative unfolding rule (R2n) is a particular case of Seki's *reduction* rule [73]. Case (P) of the rule for constrained atomic folding (R3n) is an adaptation to the case of CLP programs of Seki's folding rule [74], with the restriction that only one literal can be folded. The rule for removing useless clauses (R4u) is a variant of the rule bearing the same name presented in [62]. Finally, as already mentioned in previous sections, the rules for folding negative literals (R3n, Case N) and for contextual constraint replacement (R5n) are novel and they cannot be regarded as instances of the folding and constraint replacement rules already considered in the literature.

We would like to stress the point that, to our knowledge, the contextual specialization strategy presented in this chapter is the first technique explicitly designed for the specialization of constraint logic programs with negation. Other program specialization techniques, based upon Lloyd and Shepherdson's partial evaluation approach [50], deal with general logic programs (see, for instance, [32, 46]). Indeed, we have adapted from [46] the approach based on well-quasi orders for controlling unfolding and generalization. However, we would also like to notice that our strategy is particularly oriented to the treatment of programs with negation and constraints (see, in particular, the use of the constrained folding and contextual constraint replacement rules), and thus, it is arguable that it will produce better results when specializing such programs.

Chapter 4

Verifying CTL Properties of Infinite State Systems

Model checking is a highly successful technique for the automatic verification of properties of finite state concurrent systems [14]. In essence, it consists in: (i) modeling the concurrent 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 concurrent systems. A safety property states that 'something (bad) may never happen', while a liveness property states that 'something (good) eventually happens'.

One of the most challenging problems in the area of verification of concurrent systems, is how to extend model checking to *infinite* state concurrent systems (see, for instance, [52]). In this case, a concurrent 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, 24, 54]). 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, [55, 77, 78]). 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, [15, 18]). 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 for specifying concurrent systems and their properties, and (2) program specialization 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 $M(P_{\mathcal{K}})$ (see Section 3.1), which is equal to the unique stable model and the two-valued, well-founded model [6]. 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. 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} of the form $sat_{spec}(X) \leftarrow init(X),\ sat(X, \varphi)$, 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 (†), 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 4.4).

The specialization technique we use for program verification follows the approach based on transformation rules and strategies described in Chapter 3. We use the transformation rules of Section 3.2, which are variants of the familiar unfolding, folding, clause deletion, and constraint replacement rules. In Section 3.3 we showed that they preserve the perfect model semantics, and thus, they ensure that the equivalence (†) holds. We will also present a transformation strategy tailored to verification 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 [15, 18]. However, since generalization is applied during, and not before, the verification process, generalization may be more flexible than abstraction.

The contributions of this chapter are the following ones. (i) We have shown that the CTL properties of *concurrent systems* as defined in [76], can be expressed by using perfect models of locally stratified CLP programs. (ii) We have proposed an automatic strategy for program specialization and, in particular, a technique for generalization which makes program specialization always terminating. (iii) Finally, we have demonstrated that our technique is powerful enough to automatically verify several infinite state systems considered in the literature.

The structure of this chapter is as follows. In Section 4.1 we present an introductory example to illustrate the basic ideas of our verification method. In Section 4.2 we recall some preliminary notions concerning the CTL temporal logic. For notions concerning locally stratified constraint logic programs, see Section 3.1. In Section 4.3 we consider a class of concurrent systems and we show how CTL properties of systems in that class can be encoded by using locally stratified CLP programs. In Section 4.4 we describe our specialization strategy for verification, and we describe the technique for performing generalizations and ensuring the termination of the strategy. In Section 4.5 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 [26]. 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 4.7 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 [20, 30, 47, 67, 69].

4.1 A Preliminary Example

In this section we illustrate the basic ideas of our verification method by means of a simple example. 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 4.3. 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 4.2, $\neg EF\, null$ 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 [71], 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 introduced in Section 3.2. These transformation rules are applied in an automatic way following the verification strategy

described in Section 4.4. 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 verification 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 verification strategy (see Phase A of the strategy described in Section 4.4). 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 (†) at the beginning of the chapter), 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 $P_{Count} \cup \{\text{clause } 5\}$ [6] and sat_{spec} is defined by clause 5 only, we get $sat(1, \neg EF\, null) \in M(P_{Count})$ 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 during the unfolding process (in our case, the goal $X > 1, sat(X, EF\, null)$ which occurs both in clause 11 and clause 13) 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 4.4.

4.2 The Computational Tree Logic

In this section 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 concurrent systems. For a more detailed treatment of CTL the reader may look at [14].

The Computation Tree Logic (CTL, for short) is a temporal logic for expressing properties of the evolutions in time of concurrent 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 4.2.1. [*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.

Definition 4.2.2. [*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*, or *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 $i \geq 0$.

Notice that if R is not total then we can make it total in the following way: for all states $s \in S$, if there is no state $s' \in S$ such that $(s, s') \in R$ then we add the pair (s, s) to R.

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.

4.3 Expressing CTL Properties by Locally Stratified CLP

In this section we present the class of concurrent systems which can be verified by using our method. This class is very general, and includes the *concurrent systems* defined in [76]. But, unlike [76], 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 [20]. However, we use CLP programs with negation and the perfect model semantics, while the authors of [20] consider definite CLP programs and express temporal properties by means of least and greatest fixpoints.

For reasons of simplicity, we will consider a one-sorted [23] constraint domain. However, the extension to the many-sorted case is straightforward. For the treatment of many-sorted logic and its relation to one-sorted logic, we refer to [23].

A concurrent system is modeled by a Kripke structure $\mathcal{K}$ [14] 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 [76].)

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 $\mathcal{D} \models c_e(s)$.

For reasons of simplicity we assumed that the set of initial states and the set of states satisfying an elementary property can be specified by a constraint. However, the extension to the more general case, where these sets are specified by using a disjunction of constraints, is straightforward.

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,

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

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 4.3.1. 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 $X = 0 \vee \exists Y\ X = s(s(Y))$.

Given a constraint domain which satisfies (Property P) above, we show how to perform the rewriting needed for introducing a set of locally stratified clauses for the temporal operator AF. Indeed, we give a procedure such that, given as input a disjunction $t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ of events, produces as output a disjunction $r_1(X, Y) \vee \ldots \vee r_n(X, Y)$ of nondeterministic events which satisfies Conditions (1), (2) and (3) of the Encoding Algorithm.

Procedure *MakeME*(T, R)
Input: a disjunction $T(X, Y) \equiv t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ of events.
Output: a disjunction $R(X, Y) \equiv r_1(X, Y) \vee \ldots \vee r_n(X, Y)$ of nondeterministic events.

Let $t_1(X, Y)$ be of the form $cond(X) \wedge act(X, Y)$;
$R(X, Y) := cond(X) \wedge act(X, Y)$;
for $i = 2, \ldots, k$ **do**
  Let $t_i(X, Y)$ be a formula of the form $cond(X) \wedge act(X, Y)$;

Let $\neg cond(X)$ be partitioned into $c_1(X) \vee \ldots \vee c_m(X)$;
Let $R(X,Y)$ be of the form $r_1(X,Y) \vee \ldots \vee r_h(X,Y)$;
for $j = 1, \ldots, h$ **do**
Let $r_j(X,Y)$ be of the form $c(X) \wedge a(X,Y)$;
$R_j(X,Y) := ((c(X) \wedge cond(X)) \wedge (a(X,Y) \vee act(X,Y))) \vee$
$((c(X) \wedge c_1(X)) \wedge a(X,Y)) \vee \ldots \vee$
$((c(X) \wedge c_m(X)) \wedge a(X,Y))$
endfor
$R(X,Y) := \bigvee_{j=1}^{h} R_j(X,Y)$
endfor □

During the execution of the above procedure we arbitrarily manipulate formulas as specified by the following rules:

(1) replace a constraint c by the constraint $solve(c, X)$, where X denotes the free variables of c,

(2) replace a formula of the form $false \wedge F$ by the constraint $false$,

(3) replace a formula of the form $F \wedge false$ by the constraint $false$,

(4) replace a formula of the form $false \vee F$ by the formula F,

(5) replace a formula of the form $F \vee false$ by the formula F.

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 $length(\varphi)$ has the following inductive definition:

if $\varphi \in Elem$ then $length(\varphi) = 1$,
if φ is $\neg\varphi_1$ or $EX\,\varphi_1$ or $AF\,\varphi_1$ then $length(\varphi) = length(\varphi_1) + 1$,
if φ is $\varphi_1 \wedge \varphi_2$ or $EU(\varphi_1, \varphi_2)$ then $length(\varphi) = length(\varphi_1) + length(\varphi_2) + 1$.

We have the following theorem.

Theorem 4.3.2. [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 $e \in \mathit{Elem}$) 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 $\gamma \in \mathit{ground}(P_{\mathcal{K}})$ 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 [22] we have that $\mathcal{K}, s \models EU(\psi_1, \psi_2)$ holds iff $s \in \mathit{lfp}(\tau_{EU})$, 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:

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

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 $s \in \mathit{lfp}(\tau_{EU})$ iff $s \in \mathit{lfp}(T_{EU})$. It remains to show that for all $s \in S$, $s \in \mathit{lfp}(T_{EU})$ 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 [49, page 38], which states that the least Herbrand model of a definite logic program P is the least fixpoint of its T_P operator.

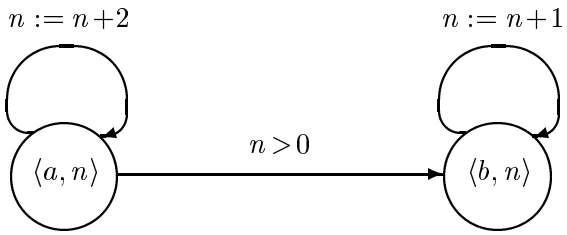

Figure 4.3.1: A simple concurrent system.

(φ is $AF\,\psi$) From [22] $\mathcal{K}, s \models AF\,\psi$ holds iff $s \in \mathit{lfp}(\tau_{AF})$, where $\tau_{AF} = \lambda I.[\psi] \cup AX^{-1}(I)$ and $AX^{-1}(I) = \{s \in S \mid \forall s' \in S \text{ if } (s, s') \in R \text{ then } s' \in I\}$. Now let us consider the operator $T_{AF} : \mathcal{P}(S) \to \mathcal{P}(S)$ defined as follows:

$$T_{AF}(I) = \{s \in S \mid sat(s, \psi) \in M(P_{\mathcal{K}}) \text{ or } \forall s' \in S \text{ if } (s, s') \in R \text{ then } s' \in I\}$$

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 $s \in \mathit{lfp}(\tau_{AF})$ iff $s \in \mathit{lfp}(T_{AF})$. It remains to show that for all $s \in S$, $s \in \mathit{lfp}(T_{AF})$ iff $sat(s, AF\,\psi) \in M(P_{\mathcal{K}})$. Again, this proof is similar to the one of Theorem 6.5 [49, 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 concurrent system modeled by a Kripke structure $\mathcal{K}$ and we apply our Encoding Algorithm for generating the corresponding program $P_{\mathcal{K}}$.

Example 4.3.3. Let us consider the concurrent system depicted in Figure 4.3.1. A state of this system is a ⟨control state, counter⟩ 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 $\mathit{init}(X_1, X_2) \equiv X_1 = a,\ X_2 = 0$, 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.)

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

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

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 4.3.1). In particular, we use the following equivalences:

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

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

$r_1(X_1, X_2, Y_1, Y_2) \equiv (X_1 = a \wedge X_2 \leq 0) \wedge (Y_1 = a \wedge Y_2 = X_2 + 2)$
$r_2(X_1, X_2, Y_1, Y_2) \equiv (X_1 = a \wedge X_2 > 0) \wedge$
$\qquad ((Y_1 = a \wedge Y_2 = X_2 + 2) \vee (Y_1 = b \wedge Y_2 = X_2))$
$r_3(X_1, X_2, Y_1, Y_2) \equiv (X_1 = b) \wedge (Y_1 = b \wedge Y_2 = X_2 + 1)$

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 $\neg AF\, neg$):

$sat(X_1, X_2, neg) \leftarrow X_2 < 0$
$sat(X_1, X_2, \neg\varphi) \leftarrow \neg sat(X_1, X_2, \varphi)$
$sat(X_1, X_2, AF\, \varphi) \leftarrow sat(X_1, X_2, \varphi)$
$sat(X_1, X_2, AF\, \varphi) \leftarrow X_1 = a,\ X_2 \leq 0,\ X_3 = X_2 + 2,\ sat(X_1, X_3, AF\, \varphi)$
$sat(X_1, X_2, AF\, \varphi) \leftarrow X_1 = a,\ X_2 > 0,\ X_3 = b,\ X_4 = X_2 + 2,$
$\qquad sat(X_1, X_4, AF\, \varphi),\ sat(X_3, X_2, AF\, \varphi)$
$sat(X_1, X_2, AF\, \varphi) \leftarrow X_1 = b,\ X_3 = X_2 + 1,\ sat(X_1, X_3, AF\, \varphi)$

4.4 The Verification Strategy

Now we present the verification strategy which we use for verifying CTL properties of concurrent systems.

Suppose that we are given a concurrent 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 4.3.2, 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 4.3). Then we apply the transformation rules of Section 3.2, according to the verification 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 4.4.3).

Our verification 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 verification strategy always terminates (see Theorem 4.4.4). 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 verification 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 4.4.1. 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 verification strategy described in Section 4.4, 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.

4.4.1 The Generalization Function

Now we present the generalization function *gen* used during Phase A of the verification strategy for introducing new clauses by using the constrained atomic definition rule.

During the Generalization Step of Phase A, given a call pattern cp of the form $(c(X), sat(X, \psi), Y)$ where $Y \subseteq X$, we introduce a new definition η of the form $newp(X) \leftarrow gen(c(X)), sat(X, \psi)$ 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 cp 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 4.1, 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 verification 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 4.5 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 4.3 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 $\mathit{Reach}(e)$ 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 $\mathcal{D} \models \forall\,(d \rightarrow e)$.

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

4.4.2 The Verification Strategy

Now we present the verification strategy which we use for verifying CTL properties of concurrent 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 4.3.

The Verification Strategy

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

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 δ_{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 $(c(X), sat(X, \psi), Y) \in CP(\Gamma)$ **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 verification 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 $\{(init(X), sat_{spec}(X))\} \cup \{(c(X), p(X)) \mid (c(X), p(X), Y) \in CP(P_A)\}$, where $CP(P_A)$ is the set of call patterns in P_A;
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 verification 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 4.4.2 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 4.4.3 and 4.4.4 below, establish the correctness and the termination of our verification strategy. We first need the following lemma.

Lemma 4.4.2. *Let P_A and P_B the output programs of Phase A and Phase B, respectively, of the verification 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 3.3.15 program P_B is stratified. □

Theorem 4.4.3. [Correctness of the Verification 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 verification 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 3.3.10)
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) P_A is stratified (by Lemma 4.4.2), (ii) $\mathbf{C} \supseteq \{(c, A) \mid \langle c, A, X\rangle \in CP(P_A)\}$, (iii) $(init(X), sat_{spec}(X)) \in \mathbf{C}$, and (iv) $\mathcal{D} \models init(s)$, then by Theorem 3.3.15 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 3.3.10, we have that $M(P_B) = M(P_{\mathcal{K},spec})$. □

Theorem 4.4.4. *The verification 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 4.4.1), (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 4.4.2), 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 4.4.5. [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 verification 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 verification strategy (see Theorem 4.4.3), we have that $sat(s, \varphi) \in M(P_{\mathcal{K}})$. Thus, by the correctness of the Encoding Algorithm (see Theorem 4.3.2), $\mathcal{K}, s \models \varphi$ holds. □

4.4.3 An Example of Application of the Verification Strategy

Let us consider the Kripke structure $\mathcal{K}$ presented in Example 4.3.3 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 verification 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 verification 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 4.4.1 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 4.4.1). 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 verification 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. $newsat2(X_1, X_2) \leftarrow X_1 = a, X_2 > 0, X_3 = X_2 + 2, X_4 = b,$
$sat(X_1, X_3, AF\ neg),\ sat(X_4, X_2, AF\ neg)$

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 $\mathbf{C}$ 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.

4.5 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 [42], (ii) the Ticket Protocol [4], and the Bounded Buffer Protocol [12].

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.

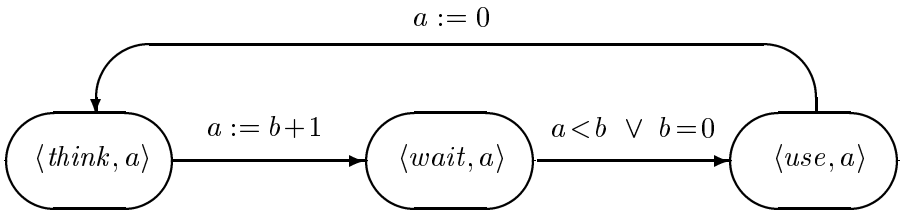

Figure 4.5.1: Process A of the Bakery Protocol.

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 [26].

4.5.1 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 4.5.1) 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):

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

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

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

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

4.5.2 The Ticket Protocol

The Ticket Protocol [4] 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 4.5.2):

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

The transitions for $(B \,|\, C)$ can be specified by a relation $R_{B|C}$, which is obtained replacing a by b in $R_{A|C}$.

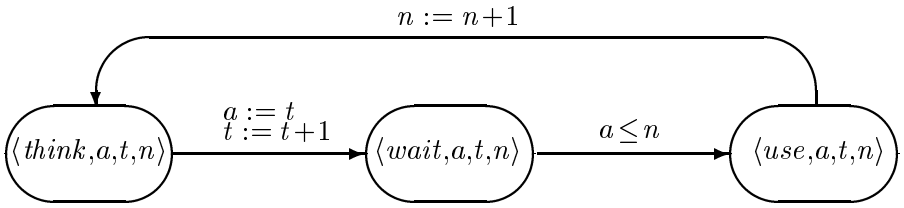

Figure 4.5.2: The Ticket Protocol: $(A \mid C)$.

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:

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

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 $c_B \in \{think, wait, use\}$.

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

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

and the transitions for $(C_i \mid B)$, 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:

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

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* is 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 $(S{-}A > sp_1{+}sp_2{-}sc_1{-}sc_2) \vee (S{-}A < sp_1{+}sp_2{-}sc_1{-}sc_2)$.

4.6 Extending the Verification Method

We now present two extensions of our Verification Method. The first extension allows us to extend the applicability of our Verification Method to a larger class of concurrent systems, by restricting the class of CTL formulas which can be used for specifying the property to be verified. The second extension allows us to prove the truth of some CTL formulas by performing a backward traversal of the state space.

Let us consider the proper subset CTL_E of CTL formulas generated by the following grammar:

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

which consists of all the CTL formulas constructed without using the operator AF.

We now modify some of the assumptions of Section 4.3 which were needed for expressing the transition relation by using constraints.

A *relational event* is a formula of the form $cond(X) \wedge act(X, Y)$, such that, $\mathcal{D} \models \forall X\, cond(X) \rightarrow \exists Y\, act(X, Y)$, where $cond(X)$ and $act(X, Y)$ are constraints whose free variables are X and $X \cup Y$, respectively.

Notice that the above definition differs from the definition of event presented in Section 4.3. Indeed, in the definition of event, we require that action act is a functional relation, that is, $\mathcal{D} \models \forall X, Y, Z\, act(X, Y) \wedge act(X, Z) \rightarrow Y = Z$, thus disallowing actions of the form $X > Y$. This condition can now be relaxed because it was only needed for introducing the clauses which specify the truth of CTL formulas of the form $AF\,\varphi$ which are not present in CTL_E.

We assume that there exists a disjunction $t_1(X, Y) \vee \ldots \vee t_k(X, Y)$ of relational events satisfying the following condition:

(TR) for all states s_1 and s_2 in S we have

$$(s_1, s_2) \in R \quad \text{iff} \quad \mathcal{D} \models t_1(s_1, s_2) \vee \ldots \vee t_k(s_1, s_2)$$

The class of concurrent systems which can be specified by using the definitions presented above allow us to specify a concurrent system $\mathcal{K} = \langle S, I, R, L\rangle$ such that, for some state $s \in S$, the set $\{s' \mid (s, s') \in R\}$ is infinite, and thus it is strictly larger than the class of concurrent systems of [76].

The following theorem states the correctness of the method presented in Section 4.3 with the modified definitions above.

Theorem 4.6.1. [Correctness of the Encoding] *Let $\mathcal{K} = \langle S, I, R, L\rangle$ be a concurrent system satisfying the conditions presented above and let $P_{\mathcal{K}}$ be a locally stratified program constructed by applying the Encoding Algorithm. Then, for all states $s \in S$ and for all formulas φ in* CTL_E*, we have that*

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

Proof. (Outline) The proof is similar to the proof of Theorem 4.3.2. □

We now describe the second extension of our Verification Method which allows us to prove the truth of CTL formulas of the form $\neg EF\, p$ or $\neg EX\, p$, where p is an elementary property, by exploring the state space backwards. This extension can be applied to the extended class of concurrent systems described above.

Let $\mathcal{K} = \langle S, I, R, L\rangle$ be a concurrent system and let p be an elementary property such that the set $S_p = \{s \in S \mid p \in L(s)\}$ of states in which p is true can be expressed by a constraint $c(X)$ over the system variables, that is, for all states s we have

$$s \in S_p \quad \text{iff} \quad \mathcal{D} \models c(s)$$

Moreover, we assume that the set I of initial states can be expressed by a constraint $init(X)$, that is, for all states s we have

$$s \in I \quad \text{iff} \quad \mathcal{D} \models init(s)$$

Let *init* be an elementary property and let $\mathcal{K}' = \langle S, S_p, R', L' \rangle$ be a concurrent system where:

- $R' = \{(s', s) \mid (s, s') \in R\}$ is the inverse of the transition relation R,
- L' is a labeling function such that, for all $s' \in S'$, $init \in L'(s)$ iff $s \in I$.

Then we have that

$\mathcal{K}', s' \models \neg EF\, init$ holds for all $s' \in S_p$ iff $\mathcal{K}, s \models \neg EF\, p$ holds for all $s \in I$

and

$\mathcal{K}', s' \models \neg EX\, init$ holds for all $s' \in S_p$ iff $\mathcal{K}, s \models \neg EX\, p$ holds for all $s \in I$

Thus, in order to prove the truth of the CTL formula $\neg EF\, p$ (respectively, $\neg EX\, p$) in system $\mathcal{K}$ we can apply our verification method to the CTL formula $\neg EF\, init$ (respectively, $\neg EX\, init$) and the system $\mathcal{K}'$.

For reasons of simplicity in this section we assumed that the set I of initial states and the set S_p can be expressed by a constraint over the system variables. However, the extension to the more general case, where the sets I and S_p are specified by using disjunctions of constraints, is straightforward.

4.7 Related Work

In recent years many logic-based techniques have been developed for automatically verifying properties of concurrent systems, the most successful of them being model checking [14]. 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 [77] 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 [67] the authors present XMC, a model checking system implemented in the tabulation-based logic programming language XSB[71]. XMC can verify temporal properties expressed in the alternation-free fragment of the μ-calculus of finite state concurrent systems specified in a CCS-like language. The XMC implementation contains many source-level optimizations 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 [20]. 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 [45] 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 coverability set, by using partial deduction algorithms.

In [58] 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 [30] 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 [69]. The verification process is carried out by proving goal equivalence in logic programs using unfold/fold based program transformation.

This chapter presents a systematic method for verifying CTL properties of infinite state concurrent systems based on a variant of the techniques developed in [27] 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 [76] 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 [42], the Ticket Protocol [4], and the Bounded Buffer Protocol [12]. 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.

Future work on the application of specialization of CLP programs for the

verification of infinite state systems will include: experimentation with different choices of constraint domains and generalization operators, and experimentation with different classes of systems and properties.

Chapter 5

Systems with an Arbitrary Number of Infinite State Processes

In this chapter we present a method for the verification of safety properties of concurrent systems which consist of finite sets of infinite state processes. This method is an enhancement of the method proposed in Chapter 4. Systems and properties are specified by using constraint logic programs, and the inference engine for verifying properties is provided by a technique based on unfold/fold program transformations. We deal with properties of finite sets of processes of arbitrary cardinality, and in order to do so, we consider constraint logic programs where the constraint theory is the *Weak Monadic Second Order Theory of k Successors*. Our verification method consists in transforming the programs that specify the properties of interest into equivalent programs where the truth of these properties can be checked by simple inspection in constant time. We present a strategy for guiding the application of the unfold/fold rules and realizing the transformations in a semiautomatic way.

5.1 Introduction

As already mentioned, model checking can be used for the verification of temporal properties of concurrent systems consisting of a *fixed* number of *finite state* processes [14]. In Chapter 4 we have presented a technique for extending model checking to concurrent systems consisting of a *fixed* number of *infinite state* processes. Recently, there have been various proposals to extend model checking for verifying properties of systems consisting of an *arbitrary* number of *infinite state* processes (see, for instance, [56, 63, 77]). The verification problem addressed by these new proposals can be formulated as follows: given

a system S_N consisting of N infinite state processes and a temporal property φ_N, prove that, for all N, the system S_N verifies property φ_N.

The main difficulty of this verification problem is that most properties of interest, such as *safety* and *liveness* properties, are undecidable for that class of concurrent systems, and thus, there cannot be any complete method for their verification. For this reason, all proposed methods resort to semiautomatic techniques, based on suitable abstractions, reduction to finite state model checking, and mathematical induction.

This chapter describes a method for verifying safety properties of systems consisting of an arbitrary number of infinite state processes. Our method avoids the use mathematical induction by abstracting away from the number N of processes actually present in the system. Indeed, this parameter does not occur in our encodings of the systems and the safety properties to be verified. These encodings are expressed as constraint logic programs, whose constraints are formulas of the *Weak Monadic Second-order Theory of k Successors*, denoted WSkS [82]. These programs will be called CLP(WSkS) programs. By using these encodings, the actual cardinality of the set of processes in the systems is not needed for the proofs of the formulas expressing the properties of interest.

Our method uses the transformation rules of Section 3.2 as inference rules for constructing proofs. Other verification methods proposed in the literature are based on CLP and/or program transformation [20, 28, 29, 47, 58, 67, 70]. However, those methods deal either with: (i) finite state systems [58, 67], or (ii) infinite state systems where the number N of infinite state processes is fixed in advance [20, 28, 29, 47], or (iii) *parameterized systems*, that is, systems consisting of an arbitrary number of finite state processes [70]. A more detailed discussion of these methods can be found in Section 5.6.

In the concurrent systems we consider, every process evolves according to its local state, called the *process state*, and the state of the other processes. Correspondingly, the whole system evolves and its state, called the *system state*, changes.

We assume that each process state consists of a pair $\langle n, s \rangle \in I\!N \times CS$, where $I\!N$ denotes the set of natural numbers and CS is a finite set. n and s are called the *counter* and the *control state* of the process, respectively. Notice that, during the evolution of the system each process may reach an infinite number of distinct states.

Since two distinct processes in a given system may have the same $\langle$counter, control state$\rangle$ pair, a system state is a *multiset* of process states.

As usual in model checking, a concurrent system is viewed as a *Kripke structure* $\mathcal{K} = \langle S, I, R, L \rangle$, where: (i) S is the set of system states, that is, the set of the multisets of $\langle$control state, counter$\rangle$ pairs, (ii) $I \subseteq S$ is a set of *initial*

system states, (iii) $R \subseteq S \times S$ is a *transition relation*, and (iv) $L : S \rightarrow \mathcal{P}(Elem)$ 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.

We also assume that for all $\langle X, Y \rangle \in R$, we have that $Y = (X - \{x\}) \cup \{y\}$ for some process states x and y, where obviously, the difference and union operations are to be understood in the multiset sense. Thus, a transition from a system state to a new system state consists in replacing a process state by a new process state. This assumption implies that: (i) the number of processes in the concurrent systems does not change over time, and (ii) the concurrent system we are modeling is asynchronous, i.e., the processes of the system do not necessarily synchronize their actions.

We will address the problem of proving safety properties of systems. A safety property is expressed by a formula of the Computational Tree Logic (see Section 4.2) of the form $\neg EF(unsafe)$, where *unsafe* is an elementary property and *EF* is a temporal operator. The meaning of any such formula is given via the satisfaction relation $\mathcal{K}, X_0 \models \neg EF(unsafe)$ which holds for a system $\mathcal{K}$ and a system state X_0 iff there is no sequence of states $X_0, X_1, \ldots, X_n$ such that: (i) for $i = 0, \ldots, n-1$, $\langle X_i, X_{i+1} \rangle \in R$ and (ii) $X_n \in unsafe$.

We may extend our method to prove more complex properties, such as those which can be expressed by using, in addition to $\neg$ and *EF*, other logical connectives and CTL temporal operators. However, for simplicity reasons, in this chapter we deal with safety properties only, and we do not consider nested temporal operators.

Now we outline our method for verifying that, for all initial system states X of a given system $\mathcal{K}$, the safety property φ holds. For the notions of locally stratified program and perfect model we refer to Section 3.1.

Verification Method.

Step 1. (*System and Property Specification*) We introduce: (i) a WSkS formula $init(X)$ which characterizes the initial system states, that is, X is an initial system state iff $init(X)$ holds, and (ii) a locally stratified CLP(WSkS) program $P_{\mathcal{K}}$ which defines a binary predicate *sat* such that for each system state X,

$$\mathcal{K}, X \models \varphi \text{ iff } sat(X, \varphi) \in M(P_{\mathcal{K}})$$

where $M(P_{\mathcal{K}})$ denotes the perfect model of the program $P_{\mathcal{K}}$.

Step 2. (*Proof Method*) We introduce a new predicate sat_{spec} defined by the CLP(WSkS) clause F: $sat_{spec}(X) \leftarrow init(X), sat(X, \varphi)$, where X is a variable. We then apply the transformation rules of Section 3.2, and from program $P_{\mathcal{K}} \cup \{F\}$ we derive a new program $P_{\mathcal{K},spec}$.
If the clause $sat_{spec}(X) \leftarrow init(X)$ occurs in $P_{\mathcal{K},spec}$ then for all initial system states X, we have that $\mathcal{K}, X \models \varphi$ holds.

The choice of the perfect model as the semantics of the program $P_{\mathcal{K}}$ requires a few words of explanation. By definition, $\mathcal{K}, X \models \neg\varphi$ holds iff $\mathcal{K}, X \models \varphi$ does not hold, and this fact can be expressed by the clause:

C: $sat(X, \neg\varphi) \leftarrow \neg\, sat(X, \varphi)$

where $\neg$ in the head of C is interpreted as a function symbol, while $\neg$ in the body of C is interpreted as negation by (finite or infinite) failure. Now, since clause C is locally stratified and the other clauses for *sat* do not contain negated atoms (see Section 5.2.2), the semantics of negation by failure is the one captured by the perfect model (recall that for locally stratified programs the perfect model is identical to the stable model and the well-founded model [6]).

This chapter is structured as follows. In Section 5.2 we describe Step 1 of our verification method and we introduce CLP(WSkS) programs, that is, constraint logic programs whose constraints are formulas in the WSkS theory. In Section 5.3 we illustrate our specification method by considering the case of a system of N processes which use the *bakery protocol* for ensuring mutual exclusion [42]. In Section 5.4 we show how Step 2 of our verification method is realized by applying a semiautomatic strategy for guiding the application of the transformation rules presented in Section 3.2 and constructing the proofs of the properties of interest. In Section 5.5, we will see our strategy in action for the verification of the N-process bakery protocol. Finally, in Section 5.6 we compare our method with the current literature in the field and we discuss possible enhancements of our method.

5.2 System and Property Specification using Weak Monadic Second Order Theories and CLP

In this section we describe Step 1 of our verification method and, in particular, we indicate how to specify a system consisting of a set of infinite state processes and how to specify its safety properties.

In order to specify a system $\mathcal{K} = \langle S, I, R, L\rangle$, we use the WSkS theory [82]. This theory is decidable [81] and it allows us to express properties of finite sets of finite strings over an alphabet of k symbols. In order to use WSkS, we represent a process state as a finite string and a system state, that is, a finite *multiset* of process states, as a finite *set* of finite strings. S is the set of system states. The set $I \subseteq S$ of initial system states is specified by a WSkS formula $init(X)$, where X is a variable ranging over finite sets of finite strings. Similarly, the transition relation R and the elementary properties in *Elem* (and, as a consequence, the labeling function L) are specified by formulas

of the form $r(X, Y)$ and $e(X)$, respectively, where X and Y range over finite sets of finite strings.

In order to specify safety properties, that is, the *sat* relation indicated at Step 1 of our verification method, we now introduce CLP programs whose constraints are WSkS formulas, denoted CLP(WSkS).

5.2.1 Constraint Logic Programs over WSkS

The syntax of WSkS is defined as follows. Let us consider a set $\Sigma = \{s_1, \ldots, s_k\}$ of k symbols, called *successors*, and a set *Ivars* of *individual variables*. An *individual term* is either a string σ or a string $x\sigma$, where $x \in$ *Ivars* and $\sigma \in \Sigma^*$, i.e., the set of the finite strings of successor symbols. By ε we denote the *empty string*.

Let us also consider a set *Svars* of *set variables* ranged over by $X, Y, \ldots$

WSkS *terms* are either individual terms or set variables.

Atomic formulas of WSkS are either (i) equalities between individual terms, written $t_1 = t_2$, or (ii) inequalities between individual terms, written $t_1 \leq t_2$, or (iii) membership atomic formulas, written $t \in X$, where t is an individual term and X is a set variable.

The *formulas* of WSkS are constructed from the atomic formulas by means of the usual logical connectives and the quantifiers over individual variables and set variables. Given any two individual terms, t_1 and t_2, we will also write $t_1 \neq t_2$ and $t_1 < t_2$, as a shorthand for $\neg(t_1 = t_2)$ and $t_1 \leq t_2 \wedge \neg(t_1 = t_2)$, respectively.

The *semantics* of WSkS formulas is defined by considering the interpretation $\mathcal{W}$ with domain Σ^* such that $=$ is interpreted as string equality, $\leq$ is interpreted as the prefix ordering on strings, and $\in$ is interpreted as membership of a string to a *finite* set of strings. We say that a closed formula φ of WSkS holds iff $\mathcal{W} \models \varphi$. The relation $\mathcal{W} \models \varphi$ is recursive [81].

A CLP(WSkS) program is a set of many-sorted first order formulas [23]. There are three sorts: *string*, *stringset*, and *tree*, interpreted as finite strings, finite sets of strings, and finite trees, respectively. We use many-sorted logic to avoid the formation of meaningless program clauses such as $p(X, s_1) \leftarrow X = s_1$, where X is a set variable of sort *stringset* and s_1 is a constant in Σ of sort *string*.

CLP(WSkS) terms are either WSkS terms or *ordinary* terms (that is, terms constructed out of variables, constants, and function symbols distinct from those used for WSkS terms). The WSkS individual terms are assigned the *string* sort, the WSkS set variables are assigned the *stringset* sort, and ordinary terms are assigned the *tree* sort. Each predicate of arity n is assigned a unique sort which consists of an n-tuple $\langle i_1, \ldots, i_n \rangle$ of sorts. For instance, the predicate $\in$ is of sort $\langle \mathit{string}, \mathit{stringset} \rangle$. We assume that CLP(WSkS) programs are

constructed by complying with the sorts of terms and predicates.

An *atom* is an atomic formula whose predicate symbol is not in $\{\leq, =, \in\}$. As usual, a *literal* is either an atom or a negated atom. A CLP(WSkS) clause is of the form $A \leftarrow c, L_1, \ldots, L_n$, where A is an atom, c is a formula of WSkS, and $L_1, \ldots, L_n$ are literals. We extended to constraint logic programs the definitions of locally stratified program and perfect model, by adapting the corresponding definitions relative to logic programs (see Section 3.1). Given a locally stratified CLP program P, $M(P)$ denotes the perfect model of P.

5.2.2 System and Property Specification Using CLP(WSkS)

Now we present our method for specifying systems and their safety properties by using CLP(WSkS). A system $\mathcal{K}$ will be specified as a tuple $\langle S, I, R, L\rangle$. Recall that a system state consists of a *multiset* of process states, that is, a multiset of pairs $\langle n, s\rangle$ where $n \in I\!N$ is a counter and $s \in CS$ is a control state. We assume that CS is the finite set $\{s_1, \ldots, s_h\}$ of symbols.

We consider the following set of successor symbols: $\Sigma = \{1, 2\} \cup CS$.

A *process state* is represented as a term of the form $1^n s 2^m$, where: (i) 1^n and 2^m are (possibly empty) strings of 1's and 2's, respectively, and (ii) s is an element of CS. For a process state $1^n s 2^m$ we have that: (i) the string 1^n represents its counter (the empty string ε represents the counter 0), and (ii) the symbol s represents its control state. The string 2^m, with different values of m, is used to allow different terms to represent the same $\langle$counter, control state$\rangle$ pair, so that a set of terms each of which is of the form $1^n s 2^m$ can be used to represent a *multiset* of process states.

Thus, a *system state* in S, which is a multiset of process states, is represented as a set of terms each of which is of the form $1^n s 2^m$.

Now we will show that process states and system states are definable by formulas in WSkS. First we need the following definitions (here and in the sequel between parentheses we write the intended meanings):

- $is\text{-}cn(x) \equiv \exists X\ ((\forall y\ y \in X \rightarrow (y{=}\varepsilon \vee \exists z\ (y{=}z\,1 \wedge z{\in}X))) \wedge x{\in}X)$
  (x is a term of the form 1^n for some $n{\geq}0$, i.e., x is a counter)

- $is\text{-}cs(x) \equiv x{=}s_1 \vee \ldots \vee x{=}s_h$
  ($x{\in}CS$, i.e., x is a control state)

Here are the WSkS formulas which define process states and system states:

- $ps(x) \equiv \exists X\ ((\forall y\ y{\in}X \rightarrow (\exists n\, \exists s\ y{=}n\,s \wedge is\text{-}cn(n) \wedge is\text{-}cs(s)) \vee$
  $\exists z\ (y{=}z\,2 \wedge z{\in}X))) \wedge x{\in}X)$
  (x is a process state, that is, a term of the form $1^n s 2^m$ for some $n, m{\geq}0$ and $s{\in}CS$)

- $ss(X) \equiv \forall x\ (x \in X \rightarrow ps(x))$ (X is a system state, that is, a set of terms of the form $1^n s 2^m$)

Now we describe the general form of the WSkS formulas which can be used for defining the transition relation of a system. We need the following two definitions:

- $cn(x, n) \equiv ps(x) \wedge \mathit{is\text{-}cn}(n) \wedge n \leq x \wedge (\forall y\ (y \leq x \wedge \mathit{is\text{-}cn}(y)) \rightarrow y \leq n)$ (n is the counter of process state x)

- $cs(x, s) \equiv ps(x) \wedge \mathit{is\text{-}cs}(s) \wedge (\exists y\ \exists z\ (y \leq x \wedge \mathit{is\text{-}cn}(z) \wedge y = z\, s)$ (s is the control state of process state x)

We recall that a transition is the replacement of a process state in a system state by a new process state. The replacement relation is defined as follows (the angle brackets $\langle, \rangle$ are used to improve readability and they are not part of the syntax of WSkS):

- $\mathit{replace}(\langle n_1, s_1\rangle, X, \langle n_2, s_2\rangle, Y) \equiv ss(X) \wedge ss(\mathrm{Y}) \wedge$
  $\exists x\,(x \in X \wedge cn(x, n_1) \wedge cs(x, s_1)) \wedge$
  $\exists y\,(y \in Y \wedge cn(y, n_2) \wedge cs(y, s_2)) \wedge$
  $\forall z\,((z \in X \wedge z \neq x) \leftrightarrow (z \in Y \wedge z \neq y))$
  ($Y = (X - \{x\}) \cup \{y\}$ for some process states $x \in X$ and $y \in Y$ such that: (i) x has counter n_1 and control state s_1 and (ii) y has counter n_2 and control state s_2)

Since any transition relation R can be viewed as the union of a finite number, say k, of relations, without loss of generality, we may assume that R is specified by a disjunction of formulas, that is, $r(X, Y) \equiv r_1(X, Y) \vee \ldots \vee r_k(X, Y)$ and, for $i = 1, \ldots, k$:

- $r_i(X, Y) \equiv \exists n_1\, \exists s_1\, \exists n_2\, \exists s_2\, (\mathit{replace}(\langle n_1, s_1\rangle, X, \langle n_2, s_2\rangle, Y) \wedge$
  $\mathit{event}_i(\langle n_1, s_1\rangle, X, \langle n_2, s_2\rangle))$

where $\mathit{event}_i(\langle n_1, s_1\rangle, X, \langle n_2, s_2\rangle)$ is any WSkS formula which specifies the transition relation of the system under consideration. Thus, we stipulate that $\langle X, Y\rangle \in R$ iff $\mathcal{W} \models r(X, Y)$.

The set I of initial system states is specified by a WSkS formula $\mathit{init}(X)$ where the set variable X is the only free variable. Similarly, an elementary property of system states is specified by a formula $e(X)$ where the set variable X is the only free variable.

Finally, the safety properties of the system $\mathcal{K}$ are specified by means of the following CLP(WSkS) program $P_{\mathcal{K}}$:

$sat(X, e_1) \leftarrow e_1(X)$
...
$sat(X, e_m) \leftarrow e_m(X)$
$sat(X, \neg\varphi) \leftarrow \neg\ sat(X, \varphi)$
$sat(X, EF(\varphi)) \leftarrow sat(X, \varphi)$
$sat(X, EF(\varphi)) \leftarrow r_1(X, Y),\ sat(Y, EF(\varphi))$
...
$sat(X, EF(\varphi)) \leftarrow r_k(X, Y),\ sat(Y, EF(\varphi))$

where $e_1, \ldots, e_m$ are the elementary properties of system states.

For any system $\mathcal{K} = \langle S, I, R, L\rangle$, program $P_{\mathcal{K}}$ is locally stratified w.r.t. the size of the second argument of *sat,* and thus, it has a unique perfect model $M(P_{\mathcal{K}})$. By Theorem 4.3.2 we have that, for any system state X in S and safety property φ of the form $\neg EF(e)$, where e is an elementary property, we have that:

$$\mathcal{K}, X \models \varphi \ \text{ iff } \ sat(X, \varphi) \in M(P_{\mathcal{K}})$$

5.3 An Example of System and Property Specification: The *N*-Process Bakery Protocol

In this section we illustrate our method for specifying systems and properties in the case of the N-process bakery protocol. This protocol ensures mutual exclusion in a system made out of N processes which use a shared resource. Mutual exclusion holds iff the shared resource is used by at most one process at a time.

Let us first give a brief description of the protocol [42]. In this protocol each process state is a ⟨counter, control state⟩ pair $\langle n, s\rangle$, where the control state s is an element of the set $CS = \{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$. The constants $\mathtt{t}$, $\mathtt{w}$, and $\mathtt{u}$ stand for *think*, *wait*, and *use*, respectively. The transition relation from a system state X, which is a multiset of process states, to a new system state Y is specified as follows (recall that the $-$ and $\cup$ operations refer to multisets).

(T1: from *think* to *wait*) if there exists a process state $\langle n, \mathtt{t}\rangle$ in X, then

$$Y = (X - \{\langle n, \mathtt{t}\rangle\}) \cup \{\langle m{+}1, \mathtt{w}\rangle\}$$

where m is the maximum value of the counters of the processes states in X,

(T2: from *wait* to *use*) if there exists a process state $\langle n, \mathtt{w}\rangle$ in X such that, for any process state $\langle m, s\rangle$ in $X - \{\langle n, \mathtt{w}\rangle\}$, either $m = 0$ or $n < m$, then

$$Y = (X - \{\langle n, \mathtt{w}\rangle\}) \cup \{\langle n, \mathtt{u}\rangle\}$$

(T3: from *use* to *think*)

$$Y = (X - \{\langle n, \mathtt{u}\rangle\}) \cup \{\langle 0, \mathtt{t}\rangle\}$$

An initial system state is any multiset of process states each of the form $\langle 0, \mathtt{t}\rangle$.

The mutual exclusion property can be specified by using the CTL formula $\neg EF(\mathit{unsafe})$, where *unsafe* is an elementary property which holds in a system state X iff there are at least two distinct process states in X with control state $\mathtt{u}$.

In order to give a formal specification of our N-process bakery protocol we use the 5 successor symbols: 1, 2, $\mathtt{t}$, $\mathtt{w}$, and $\mathtt{u}$. Thus, we consider the WS5S theory. For specifying the transition relation in cases (T1) and (T2) above, we define the following predicates *max* and *min*:

- $\mathit{max}(X,m) \equiv \exists x\ (x{\in}X \wedge \mathit{cn}(x,m)) \wedge \forall y\ \forall n\ ((y{\in}X \wedge \mathit{cn}(y,n)) \rightarrow n{\leq}m)$
  (m is the maximum counter in the system state X)

- $\mathit{min}(X,m) \equiv \exists x\ (x{\in}X \wedge \mathit{cn}(x,m)) \wedge \forall y\ \forall n\ ((y{\in}X \wedge y{\neq}x \wedge \mathit{cn}(y,n)) \rightarrow (n{=}\varepsilon \vee m{<}n))$
  (In the system state X there exists a process state x with counter m such that the counter of any process state in $X - \{x\}$ is either 0 or greater than m. Recall that the term ε denotes the counter 0.)

The transition relation between system states is defined as follows: $\langle X, Y\rangle \in R$ iff $\mathcal{W} \models \mathit{tw}(X,Y) \vee \mathit{wu}(X,Y) \vee \mathit{ut}(X,Y)$, where the predicates *tw*, *wu*, and *ut* correspond to the transition of a process from *think* to *wait*, from *wait* to *use*, and from *use* to *think*, respectively.

- $\mathit{tw}(X,Y) \equiv \exists n\, \exists m\, \mathit{replace}(\langle n, \mathtt{t}\rangle, X, \langle m\,1, \mathtt{w}\rangle, Y) \wedge \mathit{max}(X,m)$
  ($Y = (X{-}\{x\}) \cup \{y\}$, where x is a process state in X with control state $\mathtt{t}$, and y is a process with control state $\mathtt{w}$ and counter $m{+}1$ such that m is the maximum counter in X. Notice that the term $m\,1$ represents the counter $m{+}1$)

- $\mathit{wu}(X,Y) \equiv \exists n\, \mathit{replace}(\langle n, \mathtt{w}\rangle, X, \langle n, \mathtt{u}\rangle, Y) \wedge \mathit{min}(X,n)$
  ($Y = (X{-}\{x\}) \cup \{y\}$, where x is a process state in X with counter n and control state $\mathtt{w}$ such that the counter of any process state in $X{-}\{x\}$ is either 0 or greater than n, and y is a process state with counter n and control state $\mathtt{u}$)

- $\mathit{ut}(X,Y) \equiv \exists n\, \mathit{replace}(\langle n, \mathtt{u}\rangle, X, \langle \varepsilon, \mathtt{t}\rangle, Y)$
  ($Y = (X{-}\{x\}) \cup \{y\}$, where x is a process state in X with control state $\mathtt{u}$, and y is a process state with counter 0 and control state $\mathtt{t}$)

The initial and the unsafe system states are expressed by the following formulas:

- $init(X) \equiv \forall x\,(x \in X \rightarrow (cn(x,\varepsilon) \wedge cs(x,\mathtt{t})))$
  (all process states in X have counter 0 and control state $\mathtt{t}$)
- $unsafe(X) \equiv \exists x\,\exists y\,(x \in X \wedge y \in X \wedge x \neq y \wedge cs(x,\mathtt{u}) \wedge cs(y,\mathtt{u}))$
  (there exist two distinct process states in X with control state $\mathtt{u}$)

The following locally stratified CLP(WSkS) program P_{bakery} specifies the predicate sat of Step 1 of our verification method.

$sat(X, unsafe) \leftarrow unsafe(X)$
$sat(X, \neg F) \leftarrow \neg\, sat(X, F)$
$sat(X, EF(\varphi)) \leftarrow sat(X, \varphi)$
$sat(X, EF(\varphi)) \leftarrow tw(X,Y),\ sat(Y, EF(\varphi))$
$sat(X, EF(\varphi)) \leftarrow wu(X,Y),\ sat(Y, EF(\varphi))$
$sat(X, EF(\varphi)) \leftarrow ut(X,Y),\ sat(Y, EF(\varphi))$

Thus, in order to verify the safety of the bakery protocol we have to prove that, for all system states X, if $init(X)$ holds then $sat(X, \neg EF(unsafe)) \in M(P_{bakery})$.

5.4 A Strategy for Verification

In this section we show how our verification method is performed by using the unfold/fold rules of Section 3.2 for transforming CLP(WSkS) programs. In particular, we present a semiautomatic strategy for guiding the application of the transformation rules. We will see this strategy in action for the verification of a safety property of the N-process bakery protocol (see Section 5.5).

Suppose that we are given a system $\mathcal{K}$ and a safety formula φ, and we want to verify that $\mathcal{K}, X \models \varphi$ holds for all initial system states X. Suppose also that $\mathcal{K}$ and φ are specified by a CLP(WSkS) program $P_{\mathcal{K}}$ as described in Section 5.2.2. We proceed as follows. First we consider the clause:

$F.\ sat_{spec}(X) \leftarrow init(X),\ sat(X, \varphi)$

where: (i) sat_{spec} is a new predicate symbol, and (ii) $\mathcal{W} \models init(X)$ iff X is an initial system state.

Then we apply the following verification strategy which uses a *generalization function gen*. Given a WSkS formula c and a literal L which is the atom A or the negated atom $\neg A$, the function *gen* returns a definition clause $newp(v_1, \ldots, v_n) \leftarrow d, A$ such that: (i) *newp* is a new predicate symbol, (ii) $FV(d, A) = \{v_1, \ldots, v_n\}$, and (iii) $\mathcal{W} \models \forall w_1, \ldots, w_n\,(c \rightarrow d)$, where $FV(c \rightarrow d) = \{w_1, \ldots, w_n\}$.

The Verification Strategy
Input: (i) Program $P_{\mathcal{K}}$, (ii) clause F: $sat_{spec}(X) \leftarrow init(X), sat(X, \varphi)$, and (iii) generalization function gen.
Output: A program $P_{\mathcal{K},spec}$ such that for every system state X, $sat_{spec}(X) \in M(P_{\mathcal{K}} \cup \{F\})$ iff $sat_{spec}(X) \in M(P_{\mathcal{K},spec})$.

Phase A. $Defs := \{F\}$; $NewDefs := Defs$; $P := P_{\mathcal{K}}$;
while $NewDefs \neq \emptyset$ **do**

1. from $P \cup NewDefs$ derive $P \cup C_{unf}$ by unfolding once each clause in $NewDefs$;
2. from $P \cup C_{unf}$ derive $P \cup C_r$ by removing all clauses with unsatisfiable body;
3. $NewDefs := \emptyset$;
   for all clauses $\gamma \in C_r$ of the form $H \leftarrow c, G$ and for all literals L in G such that γ cannot be folded w.r.t. L using a clause in $Defs$ **do**
   $NewDefs := NewDefs \cup \{gen(c, L)\}$;
   $Defs := Defs \cup NewDefs$;
4. fold each clause in C_r w.r.t. all literals in its body and derive $P \cup C_{fld}$;
5. $P := P \cup C_{fld}$

end-while
Phase B.

1. from P derive P_u by removing all useless clauses in P;
2. from P_u derive $P_{\mathcal{K},spec}$ by unfolding the clauses in P_u w.r.t. every negative literal occurring in them.

Our verification method ends by checking whether or not clause $sat_{spec}(X) \leftarrow init(X)$ occurs in program $P_{\mathcal{K},spec}$. If it occurs, then for all initial system states X, we have that $\mathcal{K}, X \models \varphi$.

The correctness of our verification method is a consequence of the following two facts: (i) the transformation rules preserve perfect models, and (ii) perfect models are models of the completion of a program.

Theorem 5.4.1. [Correctness of the Verification Method] *Given a system $\mathcal{K}$ and a safety property φ, if $sat_{spec}(X) \leftarrow init(X)$ occurs in $P_{\mathcal{K},spec}$ then for all initial system states X, we have that $\mathcal{K}, X \models \varphi$.*

Proof. Let us assume that $sat_{spec}(X) \leftarrow init(X)$ occurs in $P_{\mathcal{K},spec}$ and let us consider an initial system state I. Thus, $\mathcal{W} \models init(I)$ and $sat_{spec}(I) \in M(P_{\mathcal{K},spec})$. By the correctness of the transformation rules (see Theorem 3.3.10), we have that $sat_{spec}(I) \in M(P_{\mathcal{K}} \cup \{F\})$. Since: (i) $M(P_{\mathcal{K}} \cup \{F\})$ is a model of the completion $comp(P_{\mathcal{K}} \cup \{F\})$, (ii) the formula $\forall X\,(sat_{spec}(X) \leftrightarrow (init(X) \wedge sat(X, \varphi))$ belongs to $comp(P_{\mathcal{K}} \cup \{F\})$, and (iii) $\mathcal{W} \models init(I)$ we have that $sat(I, \varphi) \in M(P_{\mathcal{K}} \cup \{F\})$. Now, since no sat atom in $M(P_{\mathcal{K}} \cup \{F\})$ can be inferred by using clause F, we have that $sat(I, \varphi) \in M(P_{\mathcal{K}})$ and, by Theorem 4.3.2, $\mathcal{K}, I \models \varphi$. □

The automation of our verification strategy depends on the availability of a suitable generalization function *gen*. In particular, our strategy terminates whenever the codomain of *gen* is finite. Suitable generalization functions with finite codomain can be constructed by following an approach similar to the one described in Chapter 4. More remarks on this issue will be made in Section 5.6.

5.5 Verification of the N-Process Bakery Protocol via Program Transformation

In this section we show how our verification strategy described in Section 5.4 is applied for verifying the safety of the N-process bakery protocol.

As already remarked at the end of Section 5.4, the application of our strategy can be fully automatic, provided that we are given a generalization function which introduces new definition clauses to allow folding steps (see Point 3 of the verification strategy). In particular, during the application of the strategy for the verification of the bakery protocol, we have that: (i) all formulas to be checked for applying the transformations rules are formulas of WS5S, and thus, they are decidable, and (ii) the generalization function is needed for introducing clauses d3, d9, and d16.

We start off the verification of the N-process bakery protocol by introducing the following new definition clause:

d1. $sat_{spec}(X) \leftarrow init(X),\ sat(X, \neg EF(unsafe))$

Our goal is to transform the program $P_{bakery} \cup \{\text{d1}\}$ into a program $P_{bakery,spec}$ which contains a clause of the form $sat_{spec}(X) \leftarrow init(X)$.
We start Phase A by unfolding clause 1 w.r.t. the sat atom, thereby obtaining

2. $sat_{spec}(X) \leftarrow init(X),\ \neg\ sat(X, EF(unsafe))$

The constraint $init(X)$ is satisfiable and clause 2 *cannot* be folded using the definition clause d1. Thus, we introduce the new definition clause

d3. $newp1(X) \leftarrow init(X),\ sat(X, EF(unsafe))$

By using clause d3 we fold clause 2, and we obtain

4. $sat_{spec}(X) \leftarrow init(X), \neg\, newp1(X)$

We proceed by applying the unfolding rule to the newly introduced clause d3, thereby obtaining

5. $newp1(X) \leftarrow init(X) \wedge unsafe(X)$
6. $newp1(X) \leftarrow init(X) \wedge tw(X,Y),\ sat(Y, EF(unsafe))$
7. $newp1(X) \leftarrow init(X) \wedge wu(X,Y),\ sat(Y, EF(unsafe))$
8. $newp1(X) \leftarrow init(X) \wedge ut(X,Y),\ sat(Y, EF(unsafe))$

Clauses 5, 7 and 8 are removed, because their bodies contain unsatisfiable constraints. Indeed, the following formulas hold: (i) $\forall X\ \neg(init(X) \wedge unsafe(X))$, (ii) $\forall X\ \forall Y\ \neg(init(X) \wedge wu(X,Y))$, and (iii) $\forall X\ \forall Y\ \neg(init(X) \wedge ut(X,Y))$.

Clause 6 cannot be folded using either d1 or d3, because $\forall X\, \forall Y\, (init(X) \wedge tw(X,Y) \rightarrow init(Y))$ does not hold. Thus, in order to fold clause 6, we introduce the new definition clause

d9. $newp2(X) \leftarrow c(X),\ sat(X, EF(unsafe))$

where $c(X)$ is a new constraint defined by the following WS5S formula:

$$\forall x\ (x \in X \rightarrow ((cn(x,\varepsilon) \wedge cs(x,\mathtt{t})) \vee (\exists c\ (cn(x,c) \wedge \varepsilon < c) \wedge cs(x,\mathtt{w}))))$$

denoting that every process state in the system state X is either $\langle 0, \mathtt{t}\rangle$ or $\langle c, \mathtt{w}\rangle$ for some $c > 0$. We have that $\forall X\, \forall Y\, (init(X) \wedge tw(X,Y) \rightarrow c(Y))$ holds and thus, we can fold 6 using d9. We obtain

10. $newp1(X) \leftarrow init(X) \wedge tw(X,Y),\ newp2(Y)$

By unfolding the definition clause d9 we obtain

11. $newp2(X) \leftarrow c(X) \wedge unsafe(X)$
12. $newp2(X) \leftarrow c(X) \wedge tw(X,Y),\ sat(Y, EF(unsafe))$
13. $newp2(X) \leftarrow c(X) \wedge wu(X,Y),\ sat(Y, EF(unsafe))$
14. $newp2(X) \leftarrow c(X) \wedge ut(X,Y),\ sat(Y, EF(unsafe))$

Clauses 11 and 14 have unsatisfiable constraints in their bodies and we remove them. Indeed, the following formulas hold: (i) $\forall X\ \neg(c(X) \wedge unsafe(X))$, and (ii) $\forall X\ \forall Y\ \neg(c(X) \wedge ut(X,Y))$.

We fold clause 12 by using the already introduced definition clause d9, because $\forall X\, \forall Y\, (c(X) \wedge tw(X,Y) \rightarrow c(Y))$ holds. We obtain

15. $newp2(X) \leftarrow c(X) \wedge tw(X,Y),\ newp2(Y)$

However, clause 13 cannot be folded by using a definition clause introduced so far. Thus, in order to fold clause 13, we introduce the following new definition clause

d16. $newp3(X) \leftarrow d(X),\ sat(X, EF(unsafe))$

where the constraint $d(X)$ is the WS5S formula:

$$\forall x\ (x \in X \rightarrow ((cn(x,\varepsilon) \wedge cs(x,\mathtt{t})) \vee (\exists c\ (cn(x,c) \wedge \varepsilon < c\ ) \wedge cs(x,\mathtt{w})) \vee (\exists n\ (cn(x,n) \wedge min(X,n) \wedge \varepsilon < n) \wedge cs(x,\mathtt{u})))$$

denoting that every process state in the system state X is either $\langle 0,\mathtt{t}\rangle$, or $\langle c,\mathtt{w}\rangle$ for some $c>0$, or $\langle n,\mathtt{u}\rangle$ for some $n>0$ such that no process state in X has a positive counter smaller than n. We have that $\forall X\,\forall Y\,(c(X) \wedge wu(X,Y) \rightarrow d(Y))$ holds, and thus, we can fold clause 13 using clause d16. We obtain

17. $newp2(X) \leftarrow c(X) \wedge wu(X,Y),\ \ newp3(Y)$

We now proceed by applying the unfolding rule to the definition clause d16 and we obtain

18. $newp3(X) \leftarrow d(X) \wedge unsafe(X)$
19. $newp3(X) \leftarrow d(X) \wedge tw(X,Y),\ sat(Y, EF(unsafe))$
20. $newp3(X) \leftarrow d(X) \wedge wu(X,Y),\ sat(Y, EF(unsafe))$
21. $newp3(X) \leftarrow d(X) \wedge ut(X,Y),\ sat(Y, EF(unsafe))$

We remove clause 18 because its body contains an unsatisfiable constraint because $\forall X\,\neg(d(X) \wedge unsafe(X))$ holds. Then, we fold clauses 19, 20, and 21 by using the definition clauses d16, d16, and d9, respectively. Indeed, the following three formulas hold:

$\forall X\,\forall Y\,(d(X) \wedge tw(X,Y) \rightarrow d(Y))$
$\forall X\,\forall Y\,(d(X) \wedge wu(X,Y) \rightarrow d(Y))$
$\forall X\,\forall Y\,(d(X) \wedge ut(X,Y) \rightarrow c(Y))$

We obtain

22. $newp3(X) \leftarrow d(X) \wedge tw(X,Y),\ newp3(Y)$
23. $newp3(X) \leftarrow d(X) \wedge wu(X,Y),\ newp3(Y)$
24. $newp3(X) \leftarrow d(X) \wedge ut(X,Y),\ newp2(Y)$

Since these last folding steps were performed without introducing new definition clauses, we terminate Phase A of our transformation process. The program derived so far is $P_{bakery} \cup \{4, 10, 15, 17, 22, 23, 24\}$.

Now we proceed by performing Phase B of our verification strategy. We remove the useless clauses 10, 15, 17, 22, 23, and 24 defining the predicates $newp1$, $newp2$, and $newp3$. Therefore, we derive the program $P_{bakery} \cup \{4\}$. Then we apply the unfolding rule to clause 4 w.r.t. the literal $\neg newp1(X)$, where $newp1(X)$ is a failed atom (see Point R2n of the unfolding rule). We obtain

25. $sat_{spec}(X) \leftarrow init(X)$

Thus, we derive the final program $P_{bakery,spec}$ which is $P_{bakery} \cup \{25\}$. According to our verification method, the presence of clause 25 in $P_{bakery,spec}$ proves, as desired, the mutual exclusion property for the N-process bakery protocol.

5.6 Related Work

Recently there have been several proposals of verification methods for *parameterized systems*, that is, systems consisting of an *arbitrary* number of *finite state* processes. Among them the method described in [70] is closely related to ours, in that it uses unfold/fold program transformations for generating induction proofs of safety properties for parameterized systems. However, our method differs from the method presented in [70] because we use constraint logic programs with locally stratified negation to specify concurrent systems and their properties, while [70] uses definite logic programs. Correspondingly, we use a different set of transformation rules. Moreover, we consider systems with an arbitrary number of *infinite state* processes which are more general than parameterized systems.

Now we recall the main features of some verification methods based on (constraint) logic programming, which have been recently proposed in the literature. For a more detailed discussion on these methods, see Section 4.7.
(i) The method described in [47] uses partial deduction and abstract interpretation of logic programs for verifying safety properties of infinite state systems. (ii) The method presented in [29] uses logic programs with linear arithmetic constraints and Presburger arithmetic to verify safety properties of Petri nets. (iii) The method presented in [20] uses constraint logic programs to represent infinite state systems. This method can be applied to verify CTL properties of those systems by computing approximations of least and greatest fixed points via abstract interpretation. (iv) The method proposed in [67] uses tabulation-based logic programming to efficiently verify μ-calculus properties of finite state transitions systems expressed in a CCS-like language. (v) The method described in [58] uses CLP with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking.

With respect to these methods (i)–(v), the distinctive features of our method are that: (1) we deal with systems consisting of an arbitrary number of infinite state processes, (2) we use CLP(WSkS) for their description, and (3) we apply unfold/fold program transformations for the verification of their properties.

Verification techniques for systems with an arbitrary number of infinite state processes have been presented also in the following papers.

In [56] the authors introduce a proof technique which is based on induction and model checking. Proofs are carried out by solving a finite number of model checking problems on a finite abstraction of the initial system and they are mechanically checked. The technique is illustrated by proving that the N-process bakery protocol is starvation free.

In [63] the author presents a proof of the mutual exclusion for the N-process version of the ticket protocol which is uniform w.r.t. N and it is based on the Owicki-Gries assertional method. The proof has been mechanically checked by

using the Isabelle theorem prover.

In [77] the author presents a proof of the mutual exclusion for the N-process bakery protocol. This proof is based on a combination of theorem proving, model checking, and an abstraction of the protocol itself so to reduce it to the case of two processes only.

Similarly to the techniques presented in the above three papers [56, 63, 77], each step of our verification method can be mechanized, but the construction of the whole proof requires some human guidance. However, in contrast to [56, 63, 77] in our approach the parameter N representing the number of processes is *invisible*, no explicit induction on N is performed, and no abstraction of the set of processes is needed.

More recently, in [11] the authors have presented an automated method for the verification of safety properties of parameterized systems with unbounded local data. The method, which is based on multiset rewriting and constraints, is complete for a restricted class of parameterized systems.

The verification method presented in this chapter is an enhancement of the *rules + strategies* transformation method proposed in Chapter 4 for verifying CTL properties of systems consisting of a fixed number of infinite state processes. In Chapter 4 we proved the mutual exclusion property for the 2-process bakery protocol by using CLP programs with constraints expressed by linear inequations over the reals. That proof can easily be extended to the case of any fixed number of processes by using CLP programs over the same constraint theory. Here, however, we proved the mutual exclusion property for the N-process bakery protocol, *uniformly* for any N, by using CLP programs with constraints over WSkS.

The proof of the mutual exclusion property for the N-process bakery protocol presented in Section 5.5, was done by applying under human guidance the verification strategy of Section 5.4. However, our verification method can be automated by integrating our CLP program transformation system MAP [26] with: (i) a solver for checking WSkS formulas, and (ii) suitable generalization functions for introducing new definition clauses. For Point (i) we may adapt existing implementations, such as, the MONA system [41]. Point (ii) requires further investigation but we believe that the approach presented in Chapter 4 in the case of systems consisting of a fixed number of infinite state processes can serve as a good starting point.

As already mentioned, the verification method we proposed is tailored to the verification of safety properties for *asynchronous* concurrent systems, where each transition is made by one process at a time. This limitation to asynchronous systems is a consequence of our assumption that each transition from a system state X to a new system state Y is of the form $Y = (X - \{x\}) \cup \{y\}$ for some process states x and y. In order to model *synchronous* systems, where transitions may involve more than one process at a time, we may relax this

assumption and allow transitions of the form $Y = (X - A) \cup B$ for some multisets of process states A and B. Notice that, however, these more general transitions whereby the number of processes may change over time, can be defined by WSkS formulas, and thus, it is arguable that our approach can also be used to verify properties of synchronous systems.

Bibliography

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

[2] Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. Specialization of lazy functional logic programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997* (1997), pp. 151–162.

[3] Andersen, L. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

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

[5] Apt, K. R. Introduction to logic programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 1990, pp. 493–576.

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

[7] Aravindan, C., and Dung, P. M. Partial deduction of logic programs wrt well-founded semantics. In *Algebraic and Logic Programming, ALP'92* (1992), Lecture Notes in Computer Science 632, Springer-Verlag, pp. 384–402.

[8] Baier, R., Glück, R., and Zöchling, R. Partial evaluation of numerical programs in Fortran. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)* (1994), pp. 119–132.

[9] Bensaou, N., and Guessarian, I. Transforming constraint logic programs. *Theoretical Computer Science 206* (1998), 81–125.

[10] Bossi, A., Cocco, N., and Dulli, S. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems 12*, 2 (April 1990), 253–302.

[11] Bozzano, M., and Delzanno, G. Beyond parameterized verification. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)"* (2002), Lecture Notes in Computer Science 2280, Springer, pp. 221–235.

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

[13] Burstall, R. M., and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (January 1977), 44–67.

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

[15] Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5 (1994), 1512–1542.

[16] Consel, C., and Khoo, S. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems 15*, 3 (1993), 463–493.

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

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

[19] De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41*, 2–3 (1999), 231–277.

[20] Delzanno, G., and Podelski, A. Model checking in CLP. In *5th International Conference TACAS'99* (1999), R. Cleaveland, Ed., Lecture Notes in Computer Science 1579, Springer-Verlag, pp. 223–239.

[21] Dershowitz, N., and Jouannaud, J.-P. Rewrite systems. In *Handbook of Theoretical Computer Science*, J. van Leuveen, Ed., vol. B. Elsevier, 1990, pp. 243–320.

[22] 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* (Berlin, 1981), vol. 85, Springer-Verlag, pp. 169–181.

[23] Enderton, H. *A Mathematical Introduction to Logic*. Academic Press, 1972.

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

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

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

[27] Fioravanti, F., Pettorossi, A., and Proietti, M. 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* (2001), K.-K. Lau, Ed., vol. 2042 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 125–146.

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

[29] Fribourg, L., and Olsén, H. A decompositional approach for computing least fixed-points of Datalog programs with z-counters. *Constraints 2*, 3/4 (1997), 305–335.

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

[31] Frühwirth, T. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming* (Oct. 1998), 95–138.

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

[33] Gardner, P. A., and Shepherdson, J. C. Unfold/fold transformations of logic programs. In *Computational Logic, Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT, 1991, pp. 565–583.

[34] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming* (1988), The MIT Press, pp. 1070–1080.

[35] Hickey, T. J., and Smith, D. A. Towards the partial evaluation of CLP languages. In *Proceedings ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, New Haven, CT, USA* (1991), SIGPLAN Notices, 26, 9, ACM Press, pp. 43–51.

[36] Holzbaur, C. OFAI clp(q,r) manual, edition 1.3.2. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[37] Intelligent Systems Laboratory. *SICStus Prolog 3.8.5.* Swedish Institute of Computer Science, 2000.

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

[39] Jaffar, J., Marriott, M. M. K., and Stuckey, P. The semantics of constraint logic programming. *Journal of Logic Programming 37* (1998), 1–46.

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

[41] Klarlund, N., and Møller, A. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[42] Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (1974), 453–455.

[43] LEUSCHEL, M. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997. Accessible via `http://www.ecs.soton.ac.uk/~mal`.

[44] LEUSCHEL, M. Improving homeomorphic embedding for online termination. In *Proceedings of LOPSTR'98, Manchester, UK, June 1998* (1999), P. Flener, Ed., Lecture Notes in Computer Science 1559, Springer-Verlag, pp. 199–218.

[45] LEUSCHEL, M., AND LEHMANN, H. Solving coverability problems of petri nets by partial deduction. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)* (N.Y., Sept. 20–23 2000), ACM Press, pp. 268–279.

[46] LEUSCHEL, M., MARTENS, B., AND DE SCHREYE, D. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20*, 1 (1998), 208–258.

[47] LEUSCHEL, M., AND MASSART, T. Infinite state model checking by abstract interpretation and program specialization. In *PreProceedings of LOPSTR '99, Venice, Italy* (1999), Università Ca' Foscari di Venezia, Dipartimento di Informatica, pp. 137–144.

[48] LEUSCHEL, M., AND SCHREYE, D. D. Constrained partial deduction. In *Proceedings of the 12th Workshop Logische Programmierung (WLP'97)* (Munich, Germany, September 1997), B. F. F. Bry and D. Seipel, Eds., pp. 116–126.

[49] LLOYD, J. W. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.

[50] LLOYD, J. W., AND SHEPHERDSON, J. C. Partial evaluation in logic programming. *Journal of Logic Programming 11* (1991), 217–242.

[51] MAHER, M. J. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science 110* (1993), 377–403.

[52] MANNA, Z., AND PNUELI, A. Models for reactivity. *Acta Informatica 30* (1993), 609–678.

[53] MARRIOTT, K., AND STUCKEY, P. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *POPL'93: Proceedings ACM SIGPLAN Symposium on Principles of Programming Languages* (1993), pp. 334–344.

[54] Mayr, R. Decidability of model checking with the temporal logic EF. *Theoretical Computer Science 256*, 1-2 (2001), 31–62.

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

[56] McMillan, K. L., Qadeer, S., and Saxe, J. B. Induction in compositional model checking. In *CAV 2000* (2000), Lecture Notes in Computer Science 1855, Springer-Verlag, pp. 312–327.

[57] Mendelson, E. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, Usa, Monterey, California, Usa, 1987. Third Edition.

[58] Nilsson, U., and Lübcke, J. Constraint logic programming for local and symbolic model-checking. In *CL 2000: Computational Logic* (2000), J. L. et al., Ed., no. 1861 in Lecture Notes in Artificial Intelligence, pp. 384–398.

[59] Ousterhout, J. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[60] Pettorossi, A., and Proietti, M. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19,20* (1994), 261–320.

[61] Pettorossi, A., and Proietti, M. A theory of logic program specialization and generalization for dealing with input data properties. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation* (1996), O. Danvy, R. Glück, and P. Thiemann, Eds., Lecture Notes in Computer Science 1110, Springer-Verlag, pp. 386–408.

[62] Pettorossi, A., and Proietti, M. Perfect model checking via unfold/fold transformations. In *First International Conference on Computational Logic, CL'2000, London, 24-28 July, 2000* (2000), J. L. et al., Ed., Lecture Notes in Artificial Intelligence 1861, Springer, pp. 613–628.

[63] Prensa-Nieto, L. Completeness of the owicki-gries system for parameterized parallel programs. In *Formal Methods for Parallel Programming: Theory and Applications (FMPPTA 2001)* (2001).

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

[65] Przymusinski, T. C. On the declarative semantics of stratified deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 1987, pp. 193–216.

[66] Puebla, G., and Hermenegildo, M. Abstract multiple specialization and its application to program parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41*, 2&3 (November 1999), 279–316.

[67] Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T., and Warren, D. S. Efficient model checking using tabled resolution. In *CAV '97* (1997), Lecture Notes in Computer Science 1254, Springer-Verlag, pp. 143–154.

[68] Roychoudhury, A., Kumar, K. N., Ramakrishnan, C., and Ramakrishnan, I. Proofs by program transformation. In *PreProceedings of LOPSTR '99, Venice, Italy* (1999), Università Ca' Foscari di Venezia, Dipartimento di Informatica, pp. 57–64.

[69] Roychoudhury, A., Kumar, K. N., Ramakrishnan, C., Ramakrishnan, I., and Smolka, S. 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* (2000), Springer.

[70] Roychoudhury, A., and Ramakrishnan, I. Automated inductive verification of parameterized protocols. In *CAV 2001* (2001), pp. 25–37.

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

[72] Sahlin, D. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing 12* (1993), 7–51.

[73] Seki, H. A comparative study of the well-founded and the stable model semantics: Transformation's viewpoint. In *Proceedings of the Workshop on Logic Programming and Non-monotonic Logic* (1990), Cornell University, pp. 115–123.

[74] Seki, H. Unfold/fold transformation of stratified programs. *Theoretical Computer Science 86* (1991), 107–139.

[75] Seki, H. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming 16*, 1&2 (1993), 5–23.

[76] Shankar, A. U. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys 25*, 3 (Sept. 1993), 225–262.

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

[78] Sipma, H. B., Uribe, T. E., and Manna, Z. Deductive model checking. *Formal Methods in System Design 15* (1999), 49–74.

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

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

[81] Thatcher, J. W., and Wright, J. B. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory 2* (1968), 57–82.

[82] Thomas, W. *Languages, Automata, and Logic*, vol. 3. Springer, 1997, pp. 389–455.

[83] Van Gelder, A., Ross, K., and Schlipf, J. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Sigact-Sigmod Symposium on Principles of Database Systems* (1989), ACM Press, pp. 221–230.

[84] Wrzos-Kaminska, A. Partial evaluation in constraint logic programming. In *Proceedings of the 9th International Symposium on Foundations of Intelligent Systems, Zakopane, Poland* (1996), Z. Ras and M. Michalewicz, Eds., Lecture Notes in Computer Science 1079, Springer-Verlag, pp. 98–107.

Appendix A

The MAP Transformation System

We now present the MAP system [26] which supports interactive and automated transformation of constraint logic programs.

The MAP system consists of two parts: a *transformation engine*, written in SICStus Prolog 3.8.5 [37] and a *graphical user interface* (GUI, for short) written in Tcl/Tk [59]. The interface between the transformation engine and the GUI is implemented by using SICStus's `tcltk` library.

A.1 The Transformation Engine

The transformation engine contains code for implementing (i) the transformation rules of Section 3.2 and (ii) the transformation strategies of Sections 2.5, 3.4 and 4.4. The transformation engine is located in the `src_sics/` subdirectory of the MAP installation directory and contains the following files:

- `definition_rule.pl`: contains code for implementing the constrained atomic definition rule R1;
- `unfolding_rule.pl`: contains code for implementing the positive unfolding rule R2p and the *negative unfolding* rule R2n;
- `caf.pl`: contains code for implementing the constrained atomic folding rule R3;
- `solve_clauses.pl`: contains code for implementing the constraint replacement rule R5r;
- `ccr.pl`: contains code for implementing the contextual constraint replacement rule R5n;

- `contextual_specialization.pl`: contains code which implements Phase A and Phase B of the transformation strategies of Sections 2.5, 3.4 and 4.4.

- `bup.pl`: contains code for implementing Phase C of the transformation strategies of Sections 2.5, 3.4 and 4.4. This file also contains code for the clause removal rules R4s and R4u.

The code for the transformation rules whose applicability conditions require tests for constraint satisfiability or entailment is parametric in the choice of the predicates which actually perform those tests. The implementation of the constraint solving algorithms can be found in file `solvers.pl` which contains the definitions of the following predicates. In the following, we will feel free to confuse a mathematical entity with the data structure used for representing it.

- *is_a_solver*(*Solver*) *Solver* is a ground term representing the constraint domain;

- *is_a_constraint_predicate*(*Solver*, *Pred*, *A*): *Pred* is a constraint predicate of arity *A* for *Solver*;

- *solve*(*Solver*, *C*, *X*, *D*) This predicate implements the *solve* function of Section 2.1.2 for the constraint domain *Solver*. The arguments *C*, *X* and *D* are lists representing the input constraint, the set of variables of interest, and the the output constraint, respectively.
  If *solve*(*Solver*, *C*, *X*, *D*) holds then $Solver \models \forall X((\exists Y\, C) \leftrightarrow D)$ where $Y = FV(C) - X$ and $FV(D) \subseteq FV(\exists Y\, C)$.

- *entails*(*Solver*,*C*,*D*) This predicate implements the entailment test for the constraint domain *Solver*.
  If *entails*(*Solver*,*C*,*D*) holds then $Solver \models \forall(C \rightarrow D)$.

The code for the transformation strategies of Sections 2.5 and 3.4 is independent from the definition of the predicates implementing the parameters of the strategies: the unfolding function, the widening operator for realizing clause generalization, and the well-quasi orders for controlling the unfolding process and the generalization process. The definition of these predicates can be found in the following files:

- `uf.pl`: contains the following predicates for realizing the unfolding function:

  - *is_a_uf*(*Unfold*, *S*) *Unfold* is a ground term representing an unfolding function which is compatible with the constraint domain *S*;

  - *uf*(*Unfold*, *Cl*, *Utree*, *S*, *I*) This predicate realizes the unfolding function *Unfold* such that, given a constraint domain S, a term *Cl* representing a clause of the form $H \leftarrow c, L_1, \ldots, L_n$, and a term *Utree* representing an unfolding tree, selects the positive literal L_I. This predicate must succeed if *Utree* consists of the root clause only and it must fail if there is no positive literal in the body of the considered clause.

- `widening.pl`: contains the following predicates for realizing the widening operator used in the clause generalization process:

  - *is_a_widening*(*W*, *S*) *W* is a ground term representing a widening operator which is compatible with the constraint domain *S*;
  - *widening*(*W*, *S*, *C1*, *C2*, *C3*) This predicate holds if and only if *C3* = *C1* *W* *C2*, where *W* is a widening operator and *C1*, *C2* and *C3* are constraints over *S*.

- `wqounf.pl`: contains the following predicates for realizing the well-quasi order for controlling the unfolding process:

  - *is_a_wqounf*(*Wqo*, *S*) *Wqo* is a ground term representing a well-quasi order over constrained goals which is compatible with the constraint domain *S*;
  - *embeds*(*Wqo*, *S*, *K1*, *K2*) This predicate holds if and only if *K2* *Wqo* *K1*, that is, *K2* is embedded in *K1* according to *Wqo*, where *Wqo* is a well-quasi order and *K1* and *K2* are terms representing constrained goals with constraints over *S*.

- `wqogen.pl`: contains the following predicates for realizing the well-quasi order for controlling the generalization process:

  - *is_a_wqogen*(*Wqo*, *S*) *Wqo* is a ground term representing a well-quasi order over constrained atoms which is compatible with the constraint domain *S*;
  - *embeds*(*Wqo*, *S*, *K1*, *K2*) This predicate holds if and only if *K2* *Wqo* *K1*, that is, *K2* is embedded in *K1* according to *Wqo*, where *Wqo* is a well-quasi order and *K1* and *K2* are terms representing constrained atoms with constraints over *S*.
  - *agrees_with*(*W*, *Wqo*) This predicate holds if and only if the widening operator *W* agrees with the well-quasi order *Wqo* (see Definition 2.7.1 for details).

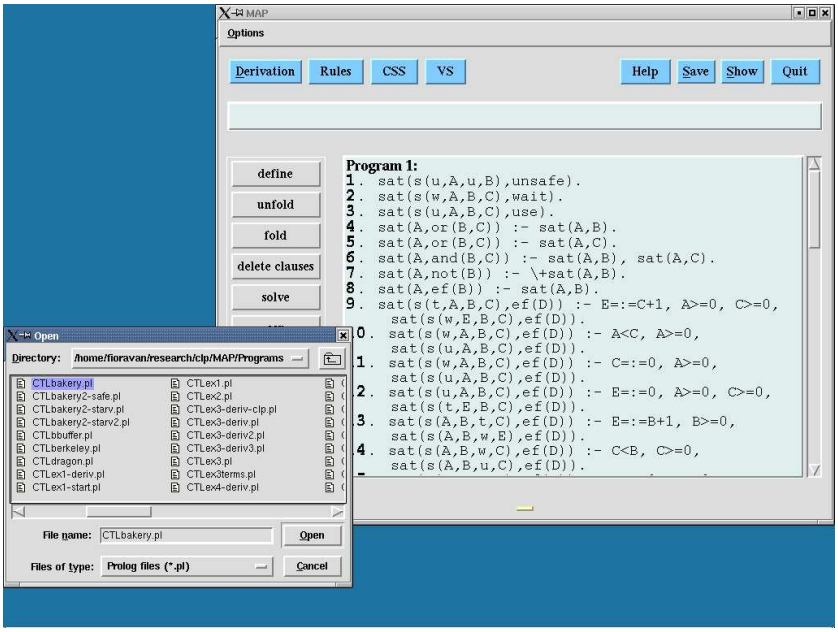

Figure A.2.1: Starting a new derivation in MAP.

The parameters for the specialization strategies can be selected through the GUI, as shown in the following section.

A.2 The Graphical User Interface

The graphical user interface provides the user a friendly way of interacting with the transformation engine by means of mouse clicks. It is implemented in the interpreted scripting language Tcl and it uses the Tk extension for managing creation, deletion and configuration of graphical objects, called *widgets*, like windows, buttons and menus.

When the MAP system starts up, the GUI creates a main window, called the *MAP* window, which contains menus and buttons for allowing the user to start a new derivation and to interact with the transformation engine. A derivation can be started by selecting one of the following items in the Derivation menu:

New opens a dialog box for selecting the file which contains the initial program of the derivation. By default, program files are located in the `Programs/` subdirectory.

Load opens a dialog box for selecting the file corresponding to a previously saved derivation. By default, derivation files are located in the `Sessions/`

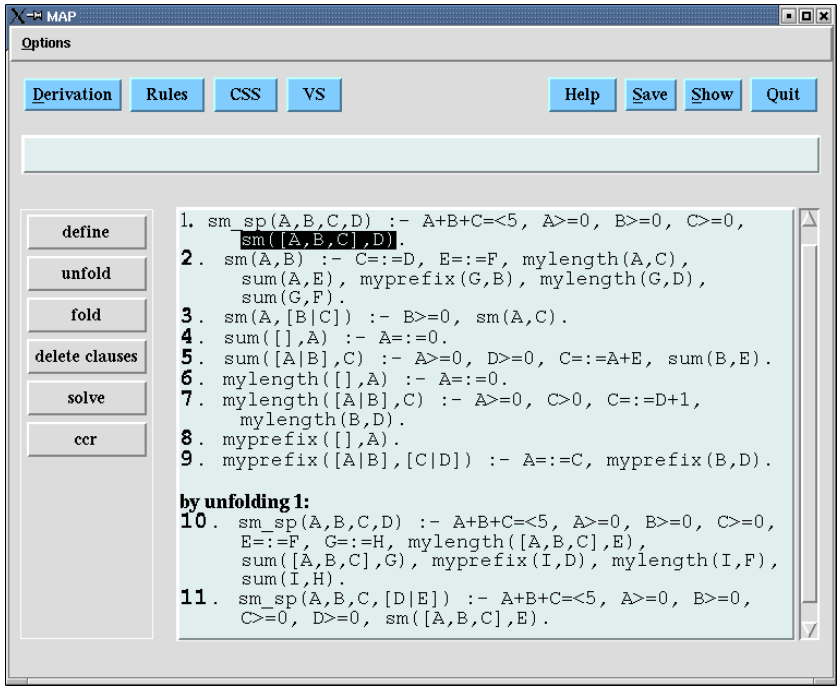

Figure A.2.2: Applying transformation rules interactively.

subdirectory.

In both cases, if no error occurs, the current program is displayed in a text area, called the *Program* window, which is contained in the *MAP* window (see Figure A.2.1).

Once a derivation is started, the user can either (i) apply the transformation rules interactively, or (ii) apply one of the automatic transformation strategies provided by the MAP system.

In interactive mode, the user selects the arguments of the transformation rule she wants to apply by clicking on them. Arguments can be literals and clauses from the *Program* window and, for applying the constrained atomic folding rule, definitions from the *Init+Defs* window. Once the arguments have been selected, the user presses the button corresponding to the chosen transformation rule (see Figure A.2.2 for an example). If the conditions of applicability of the rule are not satisfied, then an error message is displayed, otherwise the corresponding transformation is performed and the content of the *Program* window is updated by showing an informative message and the derived clauses, if any. Clauses displayed in the *Program* window are numbered and the number of a clause is in boldface style iff it belongs to the current program (see Figure A.2.2).

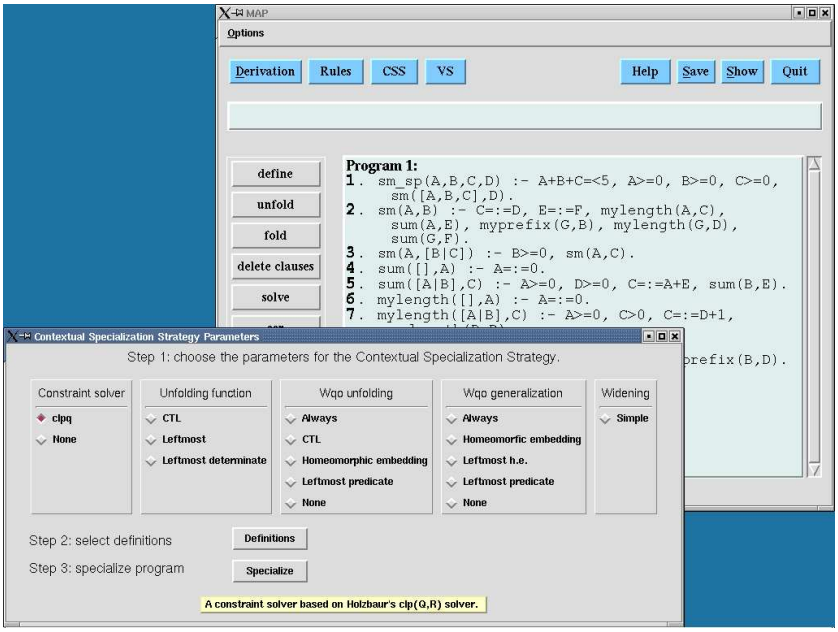

Figure A.2.3: Applying an automatic strategy.

In order to apply an automatic strategy, the user must activate the corresponding window which gives the possibility of selecting the required parameters and to start the automatic transformation process. Figure A.2.3 shows the window for applying the specialization strategy presented in Section 3.4.

Moreover, from the *MAP* window the user can activate some subsidiary windows such as (i) the *Init+Defs* window which displays the set of clauses introduced by using the constrained atomic definition rule R1, and (ii) the *History* window which contains a summary of the transformation rules which have been applied for transforming the initial program into the current one.

Appendix B

Benchmark Programs

B.1 The CLP Program *Mmod*

In this section we present the source code for program *Mmod* of Section 2.8 and for the programs generated by applying the Contextual Specialization Strategy to program *Mmod* with the constrained atom `I=0,J>=0,mmod(I,J,M)` as input.

Program *Mmod*.

```
mmod(I,J,M) :- I>=J, M=0.
mmod(I,J,M) :- I<J, I1=I+1, M=M1+L, mymod(I,L), mmod(I1,J,M1).
mymod(X,M) :- X>=0, M=X.
mymod(X,M) :- X<0, M=(-X).
```

The program obtained after Phase A.

```
mmod_pe(I,J,M) :- I=0, J=<0, M=0.
mmod_pe(I,J,M) :- I=0, J>0, I1=1, genp_pe(I1,J,M).
genp_pe(I,J,M) :- I>=J, M=0.
genp_pe(I,J,M) :- I>=0, I<J, I2=I+1, genp_pe(I2,J,M2), M=M2+I.
```

The program obtained after Phase C.

```
mmod_s_ccr(I,J,M) :- J=<0, M=0.
mmod_s_ccr(I,J,M) :- J>0, I1=1, genp_ccr(I1,J,M).
genp_ccr(I,J,M) :- I>=J, M=0.
genp_ccr(I,J,M) :- I<J, I2=I+1, genp_ccr(I2,J,M2), M=M2+I.
```

B.2 The CLP Program *SumMatch*

In this section we present the source code for program *SumMatch* of Section 2.9 and for the programs generated by applying the Contextual Specialization Strategy to program *SumMatch* with the constrained atoms

(1) `E1+E2+E3=<5,E1>=0,E2>=0,E3>=0,sm([E1,E2,E3],X)`

and

(2) `A1+A2+A3+A4+A5+A6+A7+A8+A9+A10=<5,`
`A1>=0,A2>=0,A3>=0,A4>=0,A5>=0,A6>=0,A7>=0,`
`A8>=0,A9>=0,A10>=0,sm([A1,A2,A3,A4,A5,A6,A7,A8,A9,A10],X)`

as input.

Program *SumMatch*

```
sm(P,S) :- LP=LQ,N=M,prefix(Q,S),length(P,LP),length(Q,LQ),
           sum(P,N),sum(Q,M).
sm(P,[X|Xs]) :- X>=0,sm(P,Xs).

sum([],N) :- N=0.
sum([X|Xs],N) :- X>=0,N>=0,N=X+N1,sum(Xs,N1).

length([],N) :- N=0.
length([X|Xs],N) :- X>=0,N>0,N=N1+1,length(Xs,N1).

prefix([],_).
prefix([X|Xs],[Y|Ys]) :- X=Y, prefix(Xs,Ys).
```

Let us first consider the specialization of program *SumMatch* w.r.t the constrained atom (1).

The program obtained after Phase A.

```
sm_pe3(A,B,C,[D,E,F|S]) :- A+B+C=<5,A>=0,B>=0,C>=0,
                           A+B+C=D+E+F,D>=0,E>=0,F>=0.
sm_pe3(A,B,C,[D|S]) :- D>=0,sm_pe3(A,B,C,S).
```

The program obtained after Phase C.

```
sm_c3(A,B,C,[D,E,F|S]) :- A+B+C=D+E+F,D>=0,E>=0,F>=0.
sm_c3(A,B,C,[D|S]) :- D>=0,sm_c3(A,B,C,S).
```

Let us now consider the specialization of program *SumMatch* w.r.t the constrained atom (2).

The program obtained after Phase A.

```
sm_pe10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
        [D1,D2,D3,D4,D5,D6,D7,D8,D9,D10|S]) :-
   A1+A2+A3+A4+A5+A6+A7+A8+A9+A10=<5,A1>=0,A2>=0,A3>=0,
   A4>=0,A5>=0,A6>=0,A7>=0,A8>=0,A9>=0,A10>=0,
   A1+A2+A3+A4+A5+A6+A7+A8+A9+A10=D1+D2+D3+D4+D5+D6+D7+D8+D9+D10,
   D1>=0,D2>=0,D3>=0,D4>=0,D5>=0,D6>=0,
   D7>=0,D8>=0,D9>=0,D10>=0.
sm_pe10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,[D|S]) :-
```

```
   D>=0,sm_pe10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,S).
```

The program obtained after Phase C.

```
sm_c10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
       [D1,D2,D3,D4,D5,D6,D7,D8,D9,D10|S]) :-
   A1+A2+A3+A4+A5+A6+A7+A8+A9+A10=D1+D2+D3+D4+D5+D6+D7+D8+D9+D10,
   D1>=0,D2>=0,D3>=0,D4>=0,D5>=0,D6>=0,D7>=0,D8>=0,D9>=0,D10>=0.
sm_c10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,[D|S]) :-
   D>=0,sm_c10(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,S).
```

B.3 The CLP Program *Cryptosum*

In this section we present the source code for program *Cryptosum* of Section 2.9 and for the programs generated by applying the Contextual Specialization Strategy to program *Cryptosum* with the constrained atom

```
cry([[D,N,E,S],[E,R,O,M],[Y,E,N,O,M]],[S,E,N,D,M,O,R,Y])
```

as input.

Program *Cryptosum*.

```
cry([Xs,Ys,Zs],Diff):-
    solve(Xs, Ys, Zs, [0],Carries),
    last(Xs,LXs), LXs>0,
    last(Ys,LYs), LYs>0,
    last(Zs,LZs), LZs>0,
    bits(Carries),
    gendiff(Diff,[0,1,2,3,4,5,6,7,8,9]).

solve([], [], [],  [Carry|Cs],[Carry|Cs]) :- Carry=0.
solve([], [], [Z|Zs], [Carry|Cs],OCs) :-
    Carry = Z + Carry1*10,
    solve([], [], Zs, [Carry1,Carry|Cs],OCs).
solve([], [Y], [Z|Zs], [Carry|Cs],OCs) :-
    Y + Carry = Z + Carry1*10,
    solve([], [], Zs, [Carry1,Carry|Cs],OCs).
solve([X], [], [Z|Zs], [Carry|Cs],OCs) :-
    X + Carry = Z + Carry1*10,
    solve([], [], Zs, [Carry1,Carry|Cs],OCs).
solve([X|Xs], [Y|Ys], [Z|Zs], [Carry|Cs],OCs) :-
    X + Y + Carry = Z + Carry1*10,
    solve(Xs, Ys, Zs, [Carry1,Carry|Cs],OCs).

bits([]).
```

```
bits([D|Ds]) :- member(D,[0,1]), bits(Ds).

member(X,[Y|_]) :- X=Y.
member(X,[_|Ys]) :- member(X,Ys).

last([X],X).
last([X,Y|L],Z) :- last([Y|L],Z).

gendiff([],Dom).
gendiff([X|Xs],Dom1) :- del(X,Dom1,Dom2), gendiff(Xs,Dom2).

del(X1,[X2|Xs],Xs) :- X1=X2.
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).
```

The program obtained after Phase C (which is equal to the program obtained after Phase A).

```
nc(A,B,C,D,1,F,G,H) :- D+B=H+I*10,
    C+G+I=B+J*10, B+F+J=C+K*10,
    A+K=F+9,
    A>0,
    member(K,[0,1]),
    member(J,[0,1]), member(I,[0,1]),
    ndel1(A,K1),
    ndel2(B,K1,L),
    del(C,L,M), del(D,M,N),
    ndel3(N,O),
    del(F,O,P), del(G,P,Q), del(H,Q,R).

ndel1(A,[2,3,4,5,6,7,8,9]) :- A=1.
ndel1(A,[1|B]) :- newdel1(B,A).

newdel1([3,4,5,6,7,8,9],A) :- A=2.
newdel1([2|A],B) :- newdel2(A,B).

newdel2([4,5,6,7,8,9],A) :- A=3.
newdel2([3|A],B) :- newdel3(A,B).

newdel3([5,6,7,8,9],A) :- A=4.
newdel3([4|A],B) :- newdel4(A,B).

newdel4([6,7,8,9],A) :- A=5.
newdel4([5|A],B) :- newdel5(A,B).
```

```
newdel5([7,8,9],A) :- A=6.
newdel5([6|A],B) :- newdel6(A,B).

newdel6([8,9],A) :- A=7.
newdel6([7|A],B) :- newdel7(A,B).

newdel7([9],A) :- A=8.
newdel7([8|A],B) :- newdel8(A,B).

newdel8([],A) :- A=9.

ndel2(A,B,B) :- A=0.
ndel2(A,B,[0|C]) :- newdel1_1(C,A,B).

newdel1_1(A,B,[C|A]) :- C=B.
newdel1_1([A|B],C,[A|D]) :- newdel1_1(B,C,D).

ndel3([A|B],B) :- A=1.
ndel3([A|B],[A|C]) :- ndel3(B,C).
```

B.4 The CLP Program for the 2-Process Bakery Protocol

The CLP program, written in a Prolog-like syntax, generated by the Encoding Algorithm for the concurrent system described by the bakery protocol of Section 4.5.1.

```
sat(s(u,A,u,B),unsafe).
sat(s(w,A,B,C),wait).
sat(s(u,A,B,C),use).

sat(A,or(B,C)) :- sat(A,B).
sat(A,or(B,C)) :- sat(A,C).
sat(A,and(B,C)) :- sat(A,B), sat(A,C).
sat(A,not(B)) :- \+ sat(A,B).

sat(A,ef(B)) :- sat(A,B).
sat(s(t,A,S,B),ef(C)) :- D=B+1, A>=0, B>=0,
    sat(s(w,D,S,B),ef(C)).
sat(s(w,A,S,B),ef(C)) :- A<B, A>=0, sat(s(u,A,S,B),ef(C)).
sat(s(w,A,S,B),ef(C)) :- B=0, A>=0, sat(s(u,A,S,B),ef(C)).
```

```
sat(s(u,A,S,B),ef(C)) :- D=0, A>=0, B>=0, sat(s(t,D,S,B),ef(C)).
sat(s(S,A,t,B),ef(C)) :- D=A+1, A>=0, sat(s(S,A,w,D),ef(C)).
sat(s(S,A,w,B),ef(C)) :- B<A, B>=0, sat(s(S,A,u,B),ef(C)).
sat(s(S,A,w,B),ef(C)) :- A=0, B>=0, sat(s(S,A,u,B),ef(C)).
sat(s(S,A,u,B),ef(C)) :- D=0, B>=0, A>=0, sat(s(S,A,t,D),ef(C)).

sat(A,af(B)) :- sat(A,B).
sat(s(t,T1,t,T2),af(P)) :- T3=T2+1, T4=T1+1,
    sat(s(w,T3,t,T2),af(P)), sat(s(t,T1,w,T4),af(P)).
sat(s(t,T1,u,T2),af(P)) :- T3=T2+1, T4=0,
    sat(s(w,T3,u,T2),af(P)), sat(s(t,T1,t,T4),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T2<T1,
    sat(s(w,T3,w,T2),af(P)), sat(s(t,T1,u,T2),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T1=0,
    sat(s(w,T3,w,T2),af(P)), sat(s(t,T1,u,T2),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T1>0, T1=<T2,
    sat(s(w,T3,w,T2),af(P)).
sat(s(u,T1,u,T2),af(P)) :- T3=0, T4=0,
    sat(s(t,T3,u,T2),af(P)), sat(s(u,T1,t,T4),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T2<T1,
    sat(s(t,T3,w,T2),af(P)), sat(s(u,T1,u,T2),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T1=0,
    sat(s(t,T3,w,T2),af(P)), sat(s(u,T1,u,T2),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T1>0, T1=<T2,
    sat(s(t,T3,w,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T1<T2, T1=0,
    sat(s(u,T1,w,T2),af(P)), sat(s(w,T1,u,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T1<T2, T1>0,
    sat(s(u,T1,w,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T2=0, T1=0,
    sat(s(u,T1,w,T2),af(P)), sat(s(w,T1,u,T2),af(P)).
sat(s(u,T2,t,T1),af(P)) :- T3=T2+1, T4=0,
    sat(s(u,T2,w,T3),af(P)), sat(s(t,T4,t,T1),af(P)).
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T2<T1,
    sat(s(w,T2,w,T3),af(P)), sat(s(u,T2,t,T1),af(P)).
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T1=0,
    sat(s(w,T2,w,T3),af(P)), sat(s(u,T2,t,T1),af(P)).
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T1>0, T1=<T2,
    sat(s(w,T2,w,T3),af(P)).
sat(s(u,T2,u,T1),af(P)) :- T3=0, T4=0,
    sat(s(u,T2,t,T3),af(P)), sat(s(t,T4,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T2<T1,
```

```
    sat(s(w,T2,t,T3),af(P)), sat(s(u,T2,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T1=0,
    sat(s(w,T2,t,T3),af(P)), sat(s(u,T2,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T1>0, T1=<T2,
    sat(s(w,T2,t,T3),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T1<T2, T1=0,
    sat(s(w,T2,u,T1),af(P)),
    sat(s(u,T2,w,T1),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T1<T2, T1>0,
    sat(s(w,T2,u,T1),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T2=0, T1=0,
    sat(s(w,T2,u,T1),af(P)), sat(s(u,T2,w,T1),af(P)).
```

Università degli Studi di Roma *La Sapienza*
Dottorato di Ricerca in Informatica

Collection of Ph.D. Theses

XI-00-1 Andrea Formisano. *Theory-based Resolution and Automated Set Reasoning.* Defense: March 24, 2000.

XI-00-2 Ivano Salvo. *Confluence and Expressiveness in Reduction Systems.* Defense: March 24, 2000.

XI-00-3 Marta Simeoni. *A Categorical Approach to Modularization of Graph Transformation Systems using Refinements.* Defense: March 24, 2000.

XII-01-1 Luca Forlizzi. *Algorithms and Models for Spatial Data.* Defense: June 15, 2001.

XII-01-2 Luigi Mazzucchelli. *Distributed Functional Objects.* Defense: June 15, 2001.

XII-01-3 Paolo Penna. *Resource Assignment in Wireless Networks.* Defense: June 15, 2001.

XIII-02-1 Irene Finocchi. *Hierarchical Decompositions for Visualizing Large Graphs.* Defense: February 22, 2002.

XIII-02-2 Fabio Fioravanti. *Transformation of Constraint Logic Programs for Software Specialization and Verification.* Defense: February 22, 2002.

XIII-02-3 Henry Muccini. *Software Architecture for Testing, Coordination and Views Model Checking.* Defense: February 22, 2002.