UNIVERSITÀ DEGLI STUDI "G. d'Annunzio"
CHIETI - PESCARA

SCUOLA SUPERIORE "G. d'Annunzio"

DOTTORATO DI RICERCA in SCIENZE
Ciclo XXVI

Software Verification and Synthesis using Constraints and Program Transformation

Dipartimento di Economia

Settore Scientifico Disciplinare INF/01

DOTTORANDO
Dott. Emanuele De Angelis

COORDINATORE
Prof.ssa Claudia Ceci

TUTOR
Prof. Fabio Fioravanti

Anni Accademici 2011/2013

Contents

Dedicated to the memory of
my mother Daniela

Acknowledgements

This is the hardest part to write, especially for a shy person like me. I will start with three invaluable persons: Fabio Fioravanti, Maurizio Proietti, and Alberto Pettorossi, without whom this thesis would not be possible.

Fabio, my tutor and supervisor, deserves my deep respect and admiration for the devotion he puts in his role of supervisor. He everyday places a lot of effort, and frequently spends his free time, in teaching and helping me to improve my skills. His advices and constant presence are invaluable. On a personal level, I would like to thank him for his understanding, patience, and generosity. His friendship is really important to me. He is an extraordinary person, I found more than an outstanding supervisor, Fabio is the big brother that I never had.

I am indebted to Maurizio, my supervisor, for supporting me everyday since I started to work with him during my master degree. He always believed in me, encouraging me to apply for a PhD position. I will never be able to pay him back for all the precious time he spent with me. The passion he puts in his work is a source of inspiration. In addition to his generosity and patience in sharing with me its expertise, I would like to thank him for his friendship and also for the kind understanding shown during some difficult moments of my life.

Alberto has been the supervisor of both my bachelor and master degrees. I would like to thank Alberto for having taught me so much. His devotion for students and the remarkable care he puts in his work inspire me everyday to be more hard working and pursue higher academic achievement.

I would like to thank Prof. John Gallagher and Prof. Michael Leuschel for their comments on the preliminary version of the thesis. I would also to thank Prof. Maria Chiara Meo, Prof. Claudia Ceci and Prof. Cristina Caroli Costantini who are the Director of the Computer Science Program, the Head of the PhD Program, and her predecessor, respectively.

The work presented in this thesis has been financially supported by the University of Chieti-Pescara, the Institute for Systems Analysis and Computer Science (IASI–CNR), the Italian Association for Logic Programming (GULP), the ICLP Doctoral Consortium 2012, and the National Group of Computing Science (GNCS–INDAM).

Finally, I would like to thank Valerio Senni for his friendship and my father, relatives and friends for their love.

Emanuele De Angelis
March 2014

Introduction

In the last decade formal methods applied to software production have received a renewed attention as the basis of a methodology for increasing the reliability of software artefacts (for example, source code, as well as analysis and design models) and reducing the cost of software production (for example, by reducing the time to market).

In particular, a massive effort has been made to devise automatic verification techniques, such as software model checking, for proving the correctness of programs with respect to their specifications. This thesis addresses the problem of program verification by combining and extending ideas developed in the fields of logic programming, constraint solving, abstract interpretation, and automated theorem proving. In particular, we consider program transformation of constraint logic programs to define a general verification framework which is parametric with respect to the programming language and the logic used for specifying the properties of interest. Program transformation is a software development methodology that consists in manipulating the program text by applying semantic preserving rules. It turns out to be a very flexible and general methodology by which it is possible to rapidly implement modifications of the semantics of the considered imperative language and of the logics used for expressing the properties of interest. Moreover, constraint logic programming, that is, logic programming extended with constraint solving, has been shown to be a powerful and flexible metalanguage for specifying the program syntax, the operational semantics, and the proof rules for many different programming languages and program properties.

A complementary approach to program verification is represented by program synthesis, which, starting from a formal specification of the intended behavior, has the objective of automatically deriving a program that complies with the given specification. However, program synthesis does not represent an alternative to verification in all cases. Indeed, synthesis techniques open up to the possibility of producing software artefacts that satisfy their specifications by construction, but they are much harder to put in practice, especially when scalability becomes a critical factor. This thesis addresses the problem of program synthesis by using, as done for verification of programs, techniques based on logic and constraint solving. In particular, we consider answer set programming to define a framework for automatically deriving synchronization protocols of concurrent programs. The formal specification of the protocol is given by using temporal logic formulas. Design of protocols is reduced to finding stable models, also called answer sets, of the logic program that encodes the temporal specification and the semantics of both the temporal logic and the protocol implementation language. Then, the different protocols satisfying the given specification can be derived by a simple decoding of the computed answer sets.

Overview of the Thesis

The first part of this thesis is devoted to the presentation of the verification framework [37, 38, 39, 40, 41, 42, 43, 44, 45].

In Chapter 1 we introduce the reader to the verification problem and we outline our *verification framework*. We also show, by developing a complete example taken from [44], how the verification framework can be instantiated to prove partial correctness of imperative programs written in a simple imperative programming language. This chapter is based on the work presented in [37, 38, 41, 42].

In Chapter 2 we recall basic notions of *constraint logic programming*, or CLP programs [83]. In particular, we present the *transformation rules* for CLP programs [57, 60] which will be used to define transformation strategies that realize the various steps of the verification framework presented in Chapter 1.

In Chapter 3 we show how to generate *verification conditions* for proving *partial correctness* of imperative programs written in a subset of the C programming language. We present the CLP encoding of the imperative programs and the CLP interpreter defining the semantics of the imperative language and the logic to reason about partial correctness of imperative programs. We also introduce a specialization strategy, based on the unfold/fold transformation rules, that performs the so-called removal of the interpreter which, given as input the CLP encoding of an incorrectness triple specifying the partial correctness problem, returns a set of CLP clauses expressing the verification conditions for that partial correctness problem (specified by the so-called incorrectness triple). This chapter is based on the work presented in [42, 44] and lays the foundation for the transformation techniques which will be used to check the satisfiability of the verification conditions presented in the subsequent chapters.

In Chapter 4 we show how program specialization can be used not only as a preprocessing step to generate verification conditions, but also as a means of analysis on its own [39], as an alternative to static analysis techniques of CLP programs. We extend the specialization strategy presented in Chapter 3 and we introduce *generalization operators* which are used to both discover program *invariants* and ensure the termination of the specialization process. The specialization strategy is an instance of the general specialization strategy presented in [42].

In Chapter 5 we further extend the verification method by introducing the *iterated specialization* strategy, which is based on a repeated application of program specialization [42]. During the various iterations we may apply different strategies for specializing and transforming constraint logic programs, and different operators for generalizing predicate definitions. We also perform an extended experimental evaluation of the method by using a benchmark set consisting of a significant collection of programs taken from the literature, and we compare the results with some of state-of-the-art CLP-based software model checkers [43].

In Chapter 6 we extend the verification method to perform verification of imperative programs that manipulate arrays. We propose a transformation strategy which is based on the specialization strategy for generating verification conditions and the propagation of constraints as presented in Chapters 3–5. The novel transformation strategy makes use of the constraint replacement rule that is based on an axiomatization of the array data structure. This chapter basically reports the work presented in [44], which generalizes and extends the verification method presented in [40, 41].

In Chapter 7 we present a further extension of the transformation strategy to deal with recursively defined properties. In particular, we introduce the rules of conjunctive definition introduction and conjunctive folding. We show, through an example, that this extended method can be applied in a rather systematic way, and is amenable to automation by transferring to the field of program verification many techniques developed in the field of program transformation [40].

In Chapter 8 we present the VeriMAP tool which implements the verification framework and the different strategies we have proposed. This chapter, based on the tool paper [45], also shows how to use the system by means of two examples.

In the second part of this thesis we present the synthesis framework [4, 46].

In Chapter 1 we recall basic notions of answer set programming (ASP), that is logic programming with stable model semantics. ASP is the metalanguage we use to define the synthesis framework. We also recall some basic notions of Computational Tree Logic and group theory which will be used to specify the behavioural and structural properties, of the concurrent programs to be synthesized.

In Chapter 2 we present the syntax and the semantics of concurrent programs. In particular, we introduce symmetric structures and symmetric concurrent programs which allow us to specify structural properties shared by the processes which compose the concurrent programs. We also introduce the usual behavioural properties such as mutual exclusion, starvation freedom, and bounded overtaking.

In Chapter 3 we introduce our synthesis procedure by defining the answer set programs which encode structural and behavioural properties. We present a result establishing the soundness and completeness of the synthesis procedure, and we also prove that this procedure has optimal time complexity.

In Chapter 4 we present some examples of synthesis of symmetric concurrent programs and we compare the results obtained by using different state-of-the-art answer set solvers.

In the Appendixes we show the proofs of the results presented, and the ASP source code of our synthesis procedure.

PART I

VERIFICATION

CHAPTER 1

Software Model Checking by Program Transformation

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see [90] for a recent survey).

Unfortunately, even for small programs manipulating integer variables, an exhaustive exploration of the state space generated by the execution of programs is practically infeasible, and simple properties such as safety (which essentially states that 'something bad never happens') are undecidable. Despite these limitations, software model checking techniques work in many practical cases [9, 90, 114]. Indeed, in order to tackle these issues, many software model checking techniques follow approaches based on *abstraction* [30], by which the data domain is mapped into an abstract domain so that reachability is preserved, in the sense that if a concrete configuration is reachable, then the corresponding abstract configuration is reachable. By a suitable choice of the abstract domain one can design reachability algorithms that terminate and, whenever they prove that an abstract error configuration is not reachable from any abstract initial configuration, then the program is proved to be correct. However, due to the use of abstraction, the reachability of an abstract error configuration does not necessarily imply that the program is indeed incorrect. It may happen that the abstract reachability algorithm produces a spurious counterexample, that is, a sequence of configurations leading to an abstract error configuration which does not correspond to any concrete computation. Hence, constructing such an abstraction is a critical aspect of software model checking, as it tries to meet two somewhat conflicting requirements. On one hand, in order to make the verification process of large programs viable in practice, it has to construct a model by abstracting away as many details as possible. On the other hand, it would be desirable to have a model which is as precise as possible to reduce the number of spurious error detections.

Notable abstractions are those based on convex polyhedra, that is, conjunctions of linear inequalities, also called *constraints.* In many software model checking techniques, the notion of constraint has been shown to be very effective, both for constructing models of programs and for reasoning about them [13, 16, 17, 34, 42, 49, 63, 74, 87, 86, 85, 119, 123]. Several types of constraints have been considered, such as equalities and inequalities over the booleans, the integers, the reals, the finite or infinite trees. By using constraints we can represent in a symbolic, compact way (possibly infinite) sets of values computed by programs and, more in general, sets of states reached during program executions. Then, we can use *constraint solvers*, that is, ad-hoc theorem provers, specifically designed for the various types of constraints to reason about program properties in an efficient way.

In particular, Horn clauses and constraints have been advocated by many researchers as suitable logical formalisms for the automated verification of imperative programs [16, 73, 119]. Indeed, the *verification conditions* (VC's) that express the correctness of a given program, can often be expressed as *constrained Horn clauses* [17], that is, Horn clauses extended with constraints in specific domains such as the integers or the rationals. For instance, let us consider the following C-like program:

```
int x, y, n;
while (x<n) {
  x=x+1;
  y=y+2;
}
```

Listing 1.1: Program `double`

and let us assume that we want to prove the following Hoare triple:

$\{x=0 \land y=0 \land n\geq 1\}$ `double` $\{y>x\}$.

This triple holds if the following three verification conditions are satisfiable:

1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$
2. $P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n)$
3. $P(x, y, n) \land x \geq n \rightarrow y>x$

that is, if there exists an interpretation for P such that, for all x, y and n, the three implications hold. Constraints such as the equalities and inequalities in clauses 1–3, are formulas defined in a background (possibly non-Horn) theory.

The correctness of a program is implied by the satisfiability of the verification conditions. Various methods and tools for *Satisfiability Modulo Theory* (see, for instance, [47]) prove the correctness of a given program by finding an interpretation (that is, a relation specified by constraints) that makes the verification conditions true. For instance, one such interpretation for the VC's of the program `double` is:

$$P(x,y,n) \;\equiv\; (x\!=\!0 \wedge y\!=\!0 \wedge n\!\geq\!1) \vee y\!>\!x$$

It has been noted (see, for instance, [17]) that verification conditions can also be viewed as *constraint logic programs*, or CLP programs [83]. Indeed, clauses 1 and 2 above can be considered as clauses of a CLP program over the integers, and clause 3 can be rewritten as the following *goal*:

4. $P(x,y,n) \wedge x\!\geq\! n \wedge y\!\leq\! x \;\rightarrow\; \mathit{false}$

Various verification methods based on constraint logic programming have been proposed in the literature (see, for instance, [2, 42, 49, 63, 79, 119] and the papers on which this thesis is based [39, 40, 42, 44]). These methods consist of two steps: (i) the first one is the translation of the verification task into a CLP program, and (ii) the second one is the analysis of that CLP program.

In this thesis we will show how program transformation of constraint logic programs can be used as a means to prove the satisfiability of the verification conditions. In particular, in Chapters 4 and 5 we will see that in many cases it is helpful to transform a CLP program (expressing a set of verification conditions) into an equisatisfiable program whose satisfiability is easier to show. For instance, if we propagate, according to the transformations which will see in next section and, in detail, in Chapter 5, the two constraints representing the initialization condition $(x\!=\!0 \wedge y = 0 \;\wedge n\!\geq\!1)$ and the error condition $(x\!\geq\! n \wedge y\!\leq\! x)$, then from clauses 1, 2, and 4 we derive the following new verification conditions:

5. $Q(x,y,n) \wedge x\!<\!n \wedge x\!>\!y \wedge y\!\geq\!0 \;\rightarrow\; Q(x+1, y+2, n)$
6. $Q(x,y,n) \wedge x\!\geq\!n \wedge x\!\geq\!y \wedge y\!\geq\!0 \wedge n\!\geq\!1 \;\rightarrow\; \mathit{false}$

This propagation of constraints preserves the least model, and hence, by the extension of the van Emden-Kowalski Theorem [137] to CLP programs and constrained Horn clauses, the verification conditions expressed by clauses 5 and 6 are satisfiable iff clauses 1–3 are satisfiable. Now, proving the satisfiability of clauses 5 and 6 is trivial because they are made true by simply taking $Q(x,y,n)$ to be the predicate *false*.

1.1 A Transformation-based Verification Framework

Program transformation is a software development methodology that consists in manipulating the program text by applying semantics preserving rules [24, 121, 135].

We consider program transformation techniques as a basis for building a general *verification framework* in which software model checking of programs can be performed in a very agile, effective way. The main advantage of the transformational approach to program verification over other approaches is that it allows one to construct highly configurable verification tools. Indeed, transformation-based verification techniques can easily be adapted to the various syntaxes and semantics of the programming languages under consideration and also to the various logics in which the properties of interest may be expressed.

The verification framework exploits a variety of transformation techniques (i) to generate a set of verification conditions for proving the correctness of a program, and (ii) to check the satisfiability of the generated verification conditions. In particular, it makes use of a notable program transformation technique, called *program specialization*, whose objective is the adaptation of a program to the specific context of use with the aim of deriving an equivalent program where the verification task may be easily carried out. Indeed, by following this approach, which can be regarded as a generalization of the one proposed in [119], given a program `Prog` written in a programming language L, and a property φ specified in a logic M, we can verify whether or not φ holds for `Prog` by: (i) writing the interpreter `I`, in a suitable metalanguage, defining the semantics of L and M, (ii) specializing the interpreter with respect to `Prog` and φ, thus deriving a specialized interpreter `I′`, and finally (iii) analyzing `I′`. In particular, the interpretation overhead is compiled away by specialization, and in `I′` no reference to statements of the program `Prog` is present, thereby representing the verification conditions for `Prog` in purely logical form. Moreover, since the output of a program transformation is a semantically equivalent program where the properties of interest are preserved, we can apply a *sequence* of transformations, more powerful than those needed for program specialization, thereby *refining* the analysis to the desired degree of precision.

In this thesis we develop an instance of the general transformation-based verification framework for proving partial correctness of imperative programs. We reduce the problem of verifying the partial correctness to a reachability problem, and we adopt constraint logic programming as a metalanguage for representing imperative programs, their executions, and their properties. We assume that the imperative program `Prog` is written in an imperative language L whose semantics is defined by a *transition relation*, denoted $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle c, e\rangle\!\rangle$ of a *command* c and an *environment* e. An environment e is a function that maps variable identifiers to their values. We also assume that the property φ is specified as a pair of formulas φ_{init} and φ_{error}, describing a set of *initial* and *error* configurations, respectively. Then, we address the problem of verifying whether or not, starting from any initial configuration (that is, satisfying the property φ_{init}), the execution of `Prog` eventually leads to a error configuration (that is, satisfying the property φ_{error}). This problem is formalized by defining an *incorrectness triple* of the form $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$. We say that a program `Prog`, whose free variables are assumed to be among $z_1, \ldots, z_r$, is *incorrect* with respect to φ_{init} and φ_{error} if there exist environments e_{init} and e_{halt} such that: (i) $\varphi_{init}(e_{init}(z_1), \ldots, e_{init}(z_r))$ holds, (ii) $\langle\!\langle c_0,\ e_{init}\rangle\!\rangle \Longrightarrow^* \langle\!\langle \texttt{halt},\ e_{halt}\rangle\!\rangle$, and (iii) $\varphi_{error}(e_{halt}(z_1), \ldots, e_{halt}(z_r))$ holds, where c_0 is the first command of `Prog`, `halt` is the (unique) command of `Prog` which, when executed, causes the termination of `Prog`, and $\Longrightarrow^*$ is the reflexive, transitive closure of $\Longrightarrow$. A program is said to be *correct* with respect to φ_{init} and φ_{error} if and only if it is not incorrect with respect to φ_{init} and φ_{error}. Note that this notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}$ `Prog` $\{\neg\varphi_{error}\}$.

We translate the problem of checking whether or not the program `Prog` is incorrect with respect to the properties φ_{init} and φ_{error} into the problem of checking whether or not the nullary predicate `incorrect` (standing for the predicate *false*) is a consequence of the CLP program `I` defining the interpreter for proving the partial correctness of `Prog` with respect to the properties φ_{init} and φ_{error}. Therefore, the correctness of `Prog` is defined as the negation of `incorrect`. The CLP interpreter `I` defines the following predicates:

(i) `incorrect`, which holds iff there exists an execution of the program `Prog` that leads from an initial configuration to an error configuration;

(ii) `tr` (short, for transition relation), which encodes the semantics of the imperative language L. In particular, the predicate `tr` encodes the transition relation $\Longrightarrow$ from any given configuration to the next one.

Given an incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$ and the interpreter `I`, the following four steps are performed:

Step 1: CLP Translation. The given incorrectness triple is translated into the CLP program `T`, where `Prog` is encoded as a set of CLP facts, and φ_{init} and φ_{error} are encoded by CLP predicates defined by (possibly recursive) clauses.

Step 2: Verification Conditions Generation. A set of verification conditions is generated by specializing the interpreter `I` with respect to the CLP program `T`, thereby deriving a new CLP program `V` where `tr` does not occur explicitly (in this sense the interpreter is *removed* or *compiled-away* and `V` represents a set of verification conditions for the given incorrectness triple). We have that `Prog` is incorrect with respect to φ_{init} and φ_{error} if and only if `incorrect` holds in `V`.

Step 3: Verification Conditions Transformation. By applying a sequence of program transformations the constraints encoding the properties φ_{init} and φ_{error} are propagated through the structure of the CLP program `V`. This step has the effect of modifying the structure of the program `V` and explicitly adding new constraints that denote invariants of the computation, thereby producing a new CLP program `S` such that `incorrect` holds in `V` if and only if `incorrect` holds in `S`. During this step we apply transformation techniques that are more powerful than those needed for program specialization and, in particular, for the removal of the interpreter performed by the Verification Conditions Generation step.

Step 4: Verification Conditions Analysis. The CLP program `S` is transformed into an equivalent CLP program `Q` where one of the following conditions holds: either (i) `Q` contains the fact `incorrect`, and the verification process stops reporting that `Prog` is *incorrect* with respect to φ_{init} and φ_{error}, or (ii) `Q` contains no clauses for the predicate `incorrect`, and the verification process stops reporting that `Prog` is *correct* with respect to φ_{init} and φ_{error}, or (iii) `Q` contains a clause of the form `incorrect :- G`, where `G` is not the empty goal, and the verification process returns to Step 3.

Obviously, due to undecidability limitations, it may be the case that we never get a program `Q` such that either Property (i) or Property (ii) holds, and hence the verification process does not terminate.

In order to give the reader a snapshot of our method for performing software model checking, we develop the example presented in the previous section by providing the output of each step of the verification framework in Figure 1, and the sketches of the program transformations we will present in detail in the following chapters of this thesis.

Example 1 (Proving correctness of the `double` program). Let us consider again the program `double` presented in Listing 1.1 and let us suppose that we want to prove the partial correctness of `double` with respect to $\varphi_{init}(x, y, n) =_{def}$

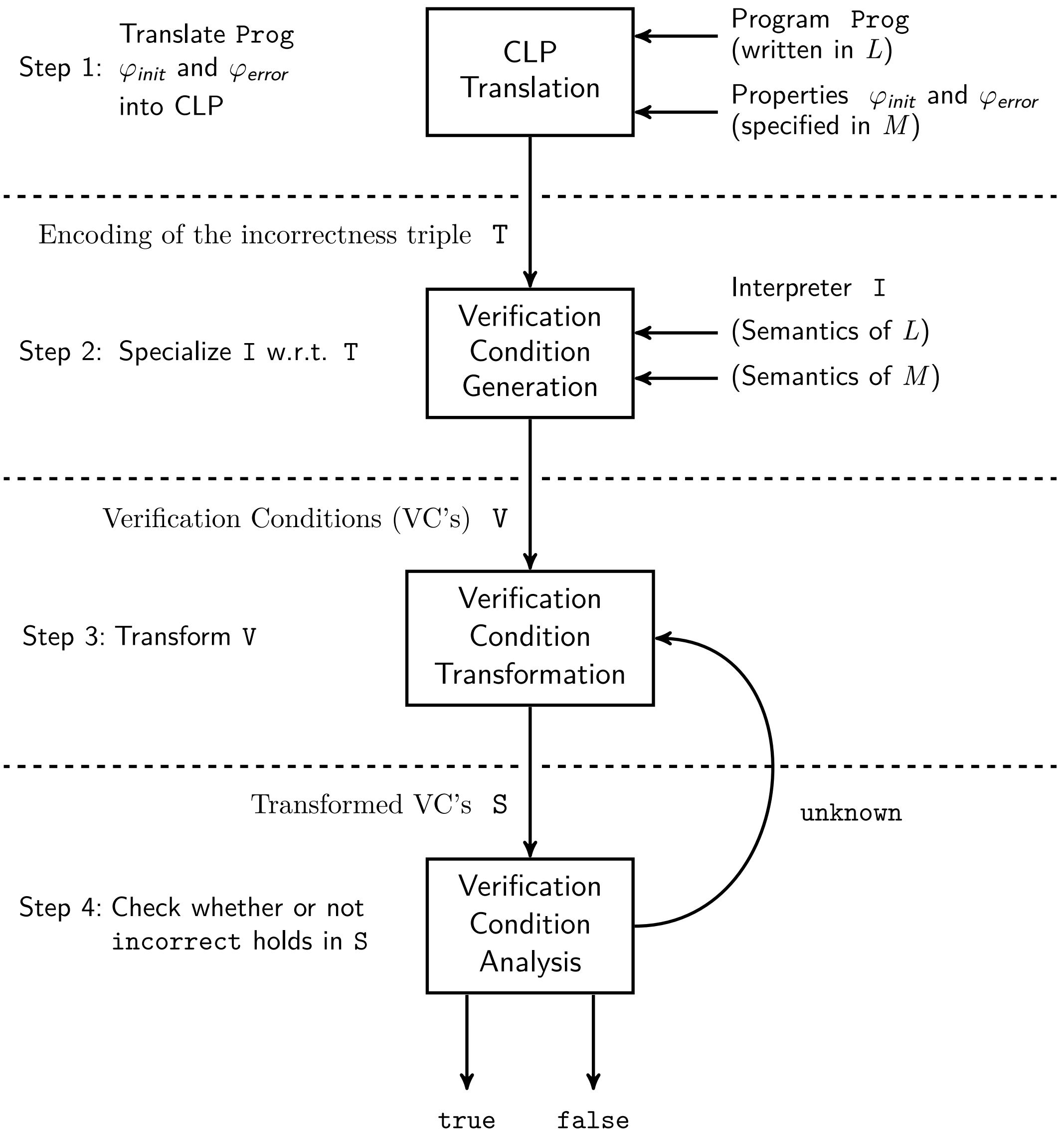

Step 1: Translate Prog φ_{init} and φ_{error} into CLP
CLP Translation
Program Prog (written in L)
Properties φ_{init} and φ_{error} (specified in M)
Encoding of the incorrectness triple T
Step 2: Specialize I w.r.t. T
Verification Condition Generation
Interpreter I
(Semantics of L)
(Semantics of M)
Verification Conditions (VC's) V
Step 3: Transform V
Verification Condition Transformation
Transformed VC's S
unknown
Step 4: Check whether or not incorrect holds in S
Verification Condition Analysis
true
false

Figure 1: Verification Framework. The verification framework relies on the subsidiary modules, represented by rectangular boxes, responsible for the operations associated with Step 1–Step 4.

$x = 0 \wedge y = 0 \wedge n \geq 1$ and $\varphi_{error}(x, y, n) =_{def} y \leq x$. That is, we want to show that, if a configuration satisfies $\varphi_{init}(x, y, n)$, by executing the program `double`, no configuration satisfying $\varphi_{error}(x, y, n)$ is reached.

In order to check whether or not $\{\!\!\{\varphi_{init}(x, y, n)\}\!\!\}$ `double` $\{\!\!\{\varphi_{error}(x, y, n)\}\!\!\}$ holds, we instantiate the verification framework of Figure 1 and we show the verification method in action on the given incorrectness triple.

We start off by executing the CLP Translation step which translates the incorrectness triple $\{\!\!\{\varphi_{init}(x, y, n)\}\!\!\}$ `double` $\{\!\!\{\varphi_{error}(x, y, n)\}\!\!\}$ into a CLP program where the predicates `at`, `phiInit`, and `phiError` represent the commands of `double`, and the initial φ_{init} and the error φ_{error} properties.

First, the commands in `double` are translated into a set of CLP facts of the form `at(L,C)`, where `L` is the label associated with the command `C`.

```
7. at(0,ite(bexp(lt(exp(id(x)),exp(id(n))),1,h)).
8. at(1,asgn(id(x),exp(plus(exp(id(x)),exp(int(1)))),2)).
9. at(2,asgn(id(y),exp(plus(exp(id(y)),exp(int(2))),0)).
10. at(h,halt).
```

The C-like commands are translated into commands of a simpler non structured core language consisting of conditionals, assignments, and jumps. For instance, the while loop of `double` is translated into the conditional (`ite`) at line 7. The first argument of `ite` represents the expression in the while loop, where: (i) `lt` represents the '`<`' operator, (ii) `bexp` and `exp` represent boolean and arithmetic expressions, respectively, and (iii) `id` represents program identifiers. The second and third argument of `ite` represent the labels of the first command occurring in the conditional branches.

Then, the initial and error properties are translated into the following CLP clauses defining the predicates `phiInit` and `phiError`, respectively.

```
11. phiInit([[int(x),X],[int(y),Y],[int(n),N]]) :- X=0,Y=0,N≥1.
12. phiError([[int(x),X],[int(y),Y],[int(n),N]]) :- Y≤X.
```

Now, in order to proceed with the Verification Conditions Generation step, we need to define the CLP interpreter `I` for proving partial correctness of C-like programs. This will be done by introducing the predicates below. The partial correctness is defined as the negation of the predicate `incorrect` specified by the following CLP clauses which encode the reachability relation between configurations:

```
13. incorrect :- initConf(X), reach(X).
14. reach(X) :- tr(X,X1), reach(X1).
```

```
15. reach(X) :- errorConf(X).
16. initConf(cf(cmd(0,C),E)) :- at(0,C), phiInit(E).
17. errorConf(cf(cmd(h,halt),E)) :- phiError(E).
```

where: (i) `initConf` encodes the initial configuration (that is, `initConf(X)` holds if `X` is a configuration consisting of the first command of the imperative program and an environment `E` where the initial property `phiInit` holds), (ii) `reach` encodes the reachability of a configuration (that is, `reach(X)` holds if an error configuration can be reached from the configuration `X`), (iii) `tr` encodes the transition relation corresponding to the operational semantics of the C-like language (see clauses 18–23 below), and (iv) `errorConf` encodes the error configurations (that is, `errorConf(X)` holds if `X` is a configuration consisting of the command `halt` and an environment `E` where the error property `phiError` holds). The predicate `tr` is defined as follows.

```
18. tr(cf(cmd(L,asgn(Id,Ae,L1)),E), cf(cmd(L1,C),Ep)) :-
        eval(Ae,E,Val), update(Id,Val,E,Ep), at(L1,C).
19. tr(cf(cmd(L,ite(Be,L1,L2)),E),cf(cmd(L1,C),E)) :-
        eval(Be,E), at(L1,C).
20. tr(cf(cmd(L,ite(Be,L1,L2)),E),cf(cmd(L2,C),E)) :-
        eval(not(Be),E), at(L2,C).
21. eval(exp(plus(Ae1,Ae2),E,Val) :- Val=Val1+Val2,
        eval(Ae1,E,Val1), eval(Ae2,E,Val2).
22. eval(exp(id(I)),E,X) :- X=Y, lookup(I,E,Y).
23. eval(exp(int(C)),E,V) :- V=C.
```

where each configuration `cf(cmd(L,C),E)` consists of (i) a labelled command `cmd(L,C)`, and (ii) an environment `E`. The environment `E` is a function that maps program variables to their values and for the program `double` is represented by the list `[[int(x),X],[int(y),Y],[int(n),N]]`. The environment is manipulated by two auxiliary predicates: (i) `update(Id,Val,E,Ep)` updates the environment `E` by binding the variable `Id` to the value `Val`, thereby constructing a new environment `Ep`, and (ii) `lookup(I,E,Y)` retrieves the value `Y` associated with the program variable `I` in the environment `E`.

In order to derive the verification conditions for the program `double`, we perform the Verification Conditions Generation step by specializing clauses 13–23 with respect to the clauses in `T` that encode the given incorrectness triple. This step is performed also in other specialization-based techniques for the program verification (see, for instance, [119]). We get the CLP program `V` consisting of the following clauses:

```
24. incorrect :- X=0, Y=0, N≥1, new1(X,Y,N).
25. new1(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, new1(X1,Y1,N).
26. new1(X,Y,N) :- X≥N, Y≤X.
```

where `new1` is a new predicate symbol automatically introduced by the specialization algorithm. We have that the satisfiability of clauses 24–26 implies the satisfiability of the corresponding VC's 1–3 of the example 1.1.

Unfortunately, it is not possible to check by direct evaluation whether or not `incorrect` is a consequence of the above CLP clauses. Indeed, the evaluation of the query `incorrect` using the standard top-down strategy enters into an infinite loop. Tabled evaluation [35] does not terminate either, as infinitely many tabled atoms are generated. Analogously, bottom-up evaluation is unable to return an answer, because infinitely many facts for `new1` should be generated for deriving that `incorrect` is not a consequence of the given CLP clauses.

Our verification method avoids the direct evaluation of the above clauses, and applies some symbolic evaluation methods based on program transformation. In particular, starting from the CLP program `V` obtained by the removal of the interpreter, we execute the Verification Conditions Transformation step, which performs further transformations, called *the propagation of the constraints*. These transformations propagate the constraints '`X=0, Y=0, N≥1`' and '`Y≤X`', characterizing the initial and error configurations, respectively.

We execute the Verification Conditions Transformation step which transforms the program `V` by propagating the constraint '`X=0, Y=0, N≥1`' and we get the following program `S`:

```
27. incorrect :- X=0, Y=0, N≥1, new2(X,Y,N).
28. new2(X,Y,N) :- X=0, Y=0, N≥1, X1=1, Y1=2, new3(X1,Y1,N).
29. new3(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, X1≥1, Y1≥2, new3(X1,Y1,N).
30. new3(X,Y,N) :- X≥N, Y≤X, Y≥0, N≥1.
```

where `new2` and `new3` are new predicate symbols introduced by the transformation algorithm.

This step produces a new program whose constraints are those occurring in the CLP program `V` enriched with new constraints derived by various constraint manipulation operations which we will see in Chapters 4 and 5. The reader may see the effect of this transformation on clauses 28–30 where we have the extra constraints (underlined in the program) '`X1=1, Y1=2`', '`X1≥1, Y1≥2`', and '`Y≥0, N≥1`', respectively.

Now, we perform the Verification Conditions Analysis step. This step is parametric with respect to the analyzer which is used (see Chapters 4 and 5 for two possible choices). Here, we use a *lightweight* analyzer which is based on a simple inspection of the program `S` and is guaranteed to terminate. Unfortunately, the analyzer is unable to check whether or not `incorrect` is a consequence of the above CLP clauses. (Note that a different analyzer, for instance, based on bottom-up evaluation, does not terminate because infinitely many facts for `new3` should be generated for deriving that `incorrect` is not a consequence of the given CLP clauses). Therefore, we proceed with a further transformation by executing again Step 3 on a new CLP program, say `R`, obtained from the CLP program `S` by reversing the direction of the state space exploration of the reachability relation `reach`. In particular, the CLP program `S`, which checks the reachability of the error configurations from the initial configurations, is transformed into the following CLP program `R`, which checks the reachability of the initial configurations from the error configurations.

```
31. incorrect :- X≥N, Y≤X, Y≥0, N≥1 new3(X,Y,N).
32. new3(X1,Y1,N) :- X=0, Y=0, N≥1, X1=1, Y1=2, new2(X,Y,N).
33. new3(X1,Y1,N) :- X<N, X1=X+1, Y1=Y+2, X1≥1,Y1≥2, new3(X,Y,N).
34. new2(X,Y,N) :- X=0, Y=0, N≥1.
```

The Verification Conditions Transformation step, applied to the CLP program `R`, propagates the constraints '`X≥N, Y≤X, Y≥0, N≥1`' (that is, the constraint '`Y≤X`' representing φ_{error} together with the constraints added by the previous execution of the Step 3). By doing so, we get the following clauses:

```
35. incorrect :- X≥N, Y≤X, Y≥0, N≥1, new4(X,Y,N).
36. new4(X1,Y1,N) :- X<N, X1=X+1, Y1=Y+2, X>Y, Y≥0, new4(X,Y,N).
```

where `new4` is a new predicate symbol introduced by the transformation algorithm, corresponding to the predicate Q of the verification condition 5, encoded by clause 36, and the verification condition 6, encoded by clause 35. Since among these final clauses 35 and 36 there are no constrained facts, their least model is empty. Thus, `incorrect` *does not hold* in the least model of the clauses 35 and 36. Then, by the correctness of the CLP encoding (Theorem 2 on page 43) and by the correctness of CLP program specialization with respect to the least model semantics (Theorem 3 on page 45 and Theorem 4 on page 67) we get, as desired, that the given imperative program `double` is correct with respect to φ_{init} and φ_{error}. □

1.2 Related Work

The use of logic programming techniques for program analysis is not novel. For instance, Datalog (the function-free fragment of logic programming) has been proposed for performing various types of program analysis such as *dataflow analysis*, *shape analysis*, and *points-to analysis* [22, 125, 139]. In order to encode the properties of interest into Datalog, all these analysis techniques make an abstraction of the program semantics. In contrast, our transformational method manipulates a CLP program which encodes the full semantics (up to a suitable level of detail) of the program to be analyzed. An advantage of our method is that the output of a transformation-based analysis is a new program which is *equivalent* to the initial one, and thus the analysis can be iterated to the desired degree of precision.

Program verification techniques based on constraint logic programming, or equivalently constrained Horn clauses, have gained increasing popularity during the last years [17, 74, 119]. As an evidence of the expressive power and flexibility of constraints we want to point out that CLP programs have been recently proposed in [16] as a common intermediate language for exchanging program properties between software verifiers. In the same line of work, [74] presents a method for the automatic synthesis of software verification tools that use proof rules for reachability and termination written in a formalism similar to Horn clauses. Moreover, the use of the CLP-based techniques allows one to enhance the reasoning capabilities within Horn clause logic because one may take advantage of the many special purpose solvers that are available for various data domains, such as integers, arrays, and other data structures.

This renewed attention to program verification techniques based on CLP can also be explained as a consequence of the development of very efficient constraint solving tools [127]. Indeed, several CLP-based verification tools are currently available. Among them, ARMC [123], Duality [113], ELDARICA [82], HSF [73], TRACER [85], μZ [81], implement reasoning techniques within CLP by following approaches based on *interpolants*, *satisfiability modulo theories*, *counterexample-guided abstraction refinement*, and *symbolic execution* of CLP programs.

As pointed out in the first section of this chapter, defining suitable program models that capture the properties of interest is a critical step in the software model checking process. In constructing such models, a software model checker may follow two dual approaches. It may start from a concrete model and then progressively abstract away some irrelevant facts. This is the approach followed by the verification method we propose in this thesis and, in a broader sense, by approaches based on symbolic execution [86]. Conversely, it may start from a coarse model and then progressively refine it by incorporating new facts. This approach is represented by the well established and widely used technique, called *Counter-Example Guided Abstraction Refinement* (CEGAR) [90], which is implemented by several software model checkers (that is, SLAM and BLAST). In particular, given a program P and a property φ, CEGAR uses an abstract model $\alpha(P)$ to check whether or not P satisfies φ. If $\alpha(P)$ satisfies φ then P satisfies φ, otherwise a counterexample, i.e., an execution which that leads to an error configuration, is produced. The counterexample is then analyzed: if it turns out to correspond to a real execution of P (*genuine* counterexample) then the program is proved to be incorrect, otherwise it has been generated due to a too coarse abstraction (*spurious* counterexample) and $\alpha(P)$ is refined to a more concrete model.

CHAPTER 2

Transformation of Constraint Logic Programs

This chapter is devoted to introduce the reader with the *transformation rules* for constraint logic programs. Transformations rules represent the basic operations for defining transformation *procedures* which, together with suitable *strategies* to control their application, will be used to perform the following steps of our verification framework: Verification Conditions Generation, Verification Conditions Transformation, and Verification Conditions Analysis.

2.1 Constraint Logic Programming

We first recall some basic notions and terminology concerning constraint logic programming. We consider CLP($\mathcal{D}$) programs, which are parametric with respect to the constraint domain $\mathcal{D}$. For more details the reader may refer to [83]. We assume that the reader is familiar with the basic notions of first order logic and logic programming [105].

2.1.1 Syntax of Constraint Logic Programs

We consider a first order language $\mathcal{L}$ generated by an infinite set $\mathcal{V}$ of *variables*, a set $\mathcal{F}$ of *function symbols* with arity $n \geq 0$, and a set $\mathcal{P}$ of *predicate symbols* with arity $n \geq 0$. We assume that $\mathcal{F}$ is the union of two disjoint sets: (i) the set $\mathcal{F}_c$ of *constraint* function symbols, and (ii) the set $\mathcal{F}_u$ of *user defined* function symbols. Similarly, we also assume that $\mathcal{P}$ is the union of two disjoint sets: (i) the set $\mathcal{P}_c$ of *constraint* predicate symbols, including `true` and `false`, and

(ii) the set $\mathcal{P}_u$ of *user defined* predicate symbols.

A *term* is either a variable in $\mathcal{V}$ or an expression of the form $\mathtt{f(t_1, \ldots, t_m)}$, where $\mathtt{f}$ is a symbol in $\mathcal{F}$ and $\mathtt{t_1, \ldots, t_m}$ are terms.

An *atomic formula* is an expression of the form $\mathtt{p(t_1, \ldots, t_m)}$, where $\mathtt{p}$ is a symbol in $\mathcal{P}$ and $\mathtt{t_1, \ldots, t_m}$ are terms.

A *formula* of $\mathcal{L}$ is either an atomic formula or a formula constructed, as usual, from already constructed formulas by means of connectives ($\neg$, $\wedge$, $\vee$, $\leftarrow$, $\leftrightarrow$) and quantifiers ($\exists$, $\forall$). We also denote '$\leftarrow$' by '`:-`' and '$\wedge$' by '`,`'.

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

An *atomic constraint* is an atomic formula $\mathtt{p(t_1, \ldots, t_m)}$, where $\mathtt{p}$ is a predicate symbol in $\mathcal{P}_c$ and $\mathtt{t_1, \ldots, t_m}$ are terms.

A *constraint* $\mathtt{c}$ is either `true`, or `false`, or an atomic constraint, or a *conjunction* $\mathtt{c_1, c_2}$ of constraints.

An *atom* is an atomic formula of the form $\mathtt{p(t_1, \ldots, t_m)}$, where $\mathtt{p}$ is a predicate symbol in $\mathcal{P}_u$ and $\mathtt{t_1, \ldots, t_m}$ are terms.

A CLP *program* P is a finite set of *clauses* of the form `A :- c, B`, where `A` is an atom, `c` is a constraint, and `B` is a (possibly empty) conjunction of atoms. `A` is called the *head* and `c, B` is called the *body* of the clause. The clause `A :- c` is called a *constrained fact*. When the constraint `c` is `true` then it is omitted and the constrained fact is called a *fact*. A *goal* is a formula of the form `:- c, B` (standing for $\mathtt{c} \wedge \mathtt{B} \rightarrow \mathtt{false}$ or, equivalently, $\neg(\mathtt{c} \wedge \mathtt{B})$). A CLP program is said to be *linear* if all its clauses are of the form `A :- c, B`, where `B` consists of at most one atom.

The *definition* of a predicate `p` in a program P is the set of all clauses of P whose head predicate is `p`.

We say that a predicate `p` *depends on* a predicate `q` in a program P if either in P there is a clause of the form `p(...) :- c, B` such that `q` occurs in `B`, or there exists a predicate `r` such that `p` depends on `r` in P and `r` depends on `q` in Π.

The set of *useless predicates* in a program P is the maximal set $\mathcal{U}$ of predicates occurring in P such that `p` is in $\mathcal{U}$ iff every clause γ with head predicate `p` is of the form $\mathtt{p(\ldots)} \leftarrow \mathtt{c} \wedge \mathtt{G_1} \wedge \mathtt{q(\ldots)} \wedge \mathtt{G_2}$ for some `q` in $\mathcal{U}$. A clause in a program P

is *useless* if the predicate of its head is a useless predicate in P.

2.1.2 Semantics of Constraint Logic Programs

We introduce a parametric (with respect to the interpretation of constraints [83, 84]) semantics of constraint logic programming. The definitions introduced in this section rely and build upon the notions introduced for logic programs [5, 105].

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

Given a formula φ where all predicate symbols belong to $\mathcal{P}_c$, we consider the satisfaction relation $\mathcal{D} \models \varphi$, which is defined as usual in the first order predicate calculus (see, for instance, [5, 105]).

Let $T_{\mathcal{D}}$ denote the set of ground terms built out of the elements of D and the function symbols $\mathcal{F}_u$ in the language of P. Given a constraint interpretation $\mathcal{D}$, an interpretation of the predicate symbols in $\mathcal{P}_u$ is called a *$\mathcal{D}$-interpretation* and is defined as follows.

A *$\mathcal{D}$-interpretation* is an interpretation with universe $T_{\mathcal{D}}$ such that: (i) it assigns to $\mathcal{F}_c$ and $\mathcal{P}_c$ the meaning according to $\mathcal{D}$, and (ii) it is the Herbrand interpretation [105] for function and predicate symbols in $\mathcal{F}_u$ and $\mathcal{P}_u$. We can identify a $\mathcal{D}$-interpretation I with the set of ground atoms (with arguments in $T_{\mathcal{D}}$) which are true in I.

We write $\mathcal{D} \models \varphi$ if φ is true in every $\mathcal{D}$-interpretation. A constraint `c` is *satisfiable* if $D \models \exists$(`c`). A constraint is *unsatisfiable* if it is not satisfiable. A constraint `c` *entails* a constraint `d`, denoted `c` $\sqsubseteq$ `d`, if $D \models \forall$(`c` $\rightarrow$ `d`).

We say that a clause of the form `H :- c, B` is satisfiable (unsatisfiable) if `c` is satisfiable (unsatisfiable).

We say that a clause of the form `H :- c, B` is *subsumed* by the constrained fact `H :- d` if `c`$\sqsubseteq$`d`.

We say that a clause `newp(X) :- c(X), Q(X)` is *more general* than a clause `newq(X) :- d(X), Q(X)` if `d(X)` $\sqsubseteq$ `c(X)`.

The semantics of a CLP program P is defined to be the *least $\mathcal{D}$-model* of P, denoted $M(P)$, that is, the least $\mathcal{D}$-interpretation that makes true every clause of P [84].

2.2 Transformation Rules

The verification method we propose is based on transformations of CLP programs that preserve the least $\mathcal{D}$-model semantics [57, 60]. The process of transforming a given CLP program P, hence deriving a new program P_t, consists in constructing a *transformation sequence* $P_0, \ldots, P_n$ of CLP programs such that:

(i) the initial program P_0 is the input program P,

(ii) the final program P_n is the output program P_t,

(iii) for $k = 0, \ldots, n-1$, the program P_{k+1} is obtained from P_k by applying one of the transformation rules presented in the following Definitions 1–5.

We assume that in every clause the head arguments denoting elements of $\mathcal{D}$ are distinct variables. This assumption is needed to guarantee that the use of unification in the unfolding rule preserves the least $\mathcal{D}$-model semantics. The transformation rules we present are variants of the unfold/fold rules considered in the literature for transforming (constraint) logic programs [11, 57, 60, 106, 131, 135].

We start by presenting the *unfolding* rule which essentially consists in replacing an atom `A` occurring in the body of a clause in P_k by the bodies of the clauses which define `A` in P_0.

Definition 1 (Unfolding). Given a clause C in P_k of the form `H :- c,L,A,R`, where `H` and `A` are atoms, `c` is a constraint, and `L` and `R` are (possibly empty) conjunctions of atoms, let $\{\mathtt{K}_i \texttt{ :- } \mathtt{c}_i\texttt{,}\mathtt{B}_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart) clauses in program P_0 such that, for $i = 1, \ldots, m$, `A` is unifiable with $\mathtt{K}_i$ via the most general unifier ϑ_i and $(\mathtt{c}\texttt{,}\mathtt{c}_i)\,\vartheta_i$ is satisfiable. We define the following function *Unf*:

$$\mathit{Unf}(C, \mathtt{A}) = \{(\mathtt{H} \texttt{ :- } \mathtt{c}\texttt{,}\mathtt{c}_i\texttt{,}\mathtt{L}\texttt{,}\mathtt{B}_i\texttt{,}\mathtt{R})\,\vartheta_i \mid i = 1, \ldots, m\}$$

Each clause in $\mathit{Unf}(C, \mathtt{A})$ is said to be derived by *unfolding* C *with respect to* `A`. By unfolding C in P_k w.r.t. `A` we derive the program $P_{k+1} = (P_k - \{C\}) \cup \mathit{Unf}(C, \mathtt{A})$.

We present a rule to remove: (i) unsatisfiable clauses, (ii) subsumed clauses, and (iii) useless clauses.

Definition 2 (Clause removal). Let C be a clause in P_k of the form `H :- c, A`. By *clause removal* we derive the program $P_{k+1} = P_k - \{C\}$ if either (i) the constraint `c` is unsatisfiable, or (ii) C is subsumed by a clause occurring in $P_k - \{C\}$, or (iii) C is useless in P_k.

We present a rule to replace a constraint occurring in the body of a clause by an equivalent disjunction of constraints.

Definition 3 (Constraint replacement). Given a clause C of the form: `H :- c0, B`, and some constraints $\mathtt{c_1}, \ldots, \mathtt{c_n}$ such that
$\mathcal{A} \models \forall\,((\exists \mathtt{X_0}\, \mathtt{c_0}) \leftrightarrow (\exists \mathtt{X_1}\, \mathtt{c_1} \vee \ldots \vee \exists \mathtt{X_n}\, \mathtt{c_n}))$
where, for $i=0, \ldots, \mathtt{n}$, $\mathtt{X_i} = \mathit{vars}(\mathtt{c_i}) - \mathit{vars}(\mathtt{H}, \mathtt{B})$, then, we derive $\mathtt{n}$ clauses $C_1, \ldots, C_{\mathtt{n}}$ obtained by replacing in the body of C the constraint $\mathtt{c_0}$ by the $\mathtt{n}$ constraints $\mathtt{c_1}, \ldots, \mathtt{c_n}$, respectively. By constraint replacement we derive the program $P_{k+1} = (P_k - \{C\}) \cup \{C_1, \ldots, C_{\mathtt{n}}\}$.

A transformation sequence $P_0, \ldots, P_k$ introduces a set Defs_k of new *predicate definitions* through the following rule.

Definition 4 (Definition introduction). A *definition* is a clause C of the form: `newp(X) :- c, A`, where: (i) `newp` is a fresh new predicate symbol not occurring in $\{P_0, \ldots, P_k\}$, (ii) `X` is the tuple of distinct variables occurring in `A`, (iii) `c` is a constraint, and (iv) every predicate symbol occurring in `A` also occurs in P_0. By definition introduction we derive the program $P_{k+1} = P_k \cup \{C\}$.

The *folding* rule consists in replacing an instance of the body of the definition of a predicate by the corresponding head.

Definition 5 (Folding). Given a clause E: `H :- e, L, A, R` in P_k and a clause D: `K :- d, D` in Defs_k. Suppose that, for some substitution ϑ, (i) $\mathtt{A} = \mathtt{D}\,\vartheta$, and (ii) $e \sqsubseteq d\vartheta$. Then by *folding E using D* we derive the clause F: $\mathtt{H :- e, L, K}\,\vartheta\mathtt{, R}$ and the program $P_{k+1} = (P_k - \{E\}) \cup \{F\}$.

The following Theorem states that, under suitable conditions, the transformation rules preserve the least $\mathcal{D}$-model of the initial program P_0.

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

Proof. The result is proved in [57, 61]. □

CHAPTER 3

Generating Verification Conditions by Specializing Interpreters

Verification conditions can be automatically generated either from a formal specification of the operational semantics of programs [119] or from the proof rules that formalize program correctness in an axiomatic way [74].

We address the problem of generating verification conditions by following an approach based on *program specialization* techniques. Program specialization is a program transformation technique which, given a program P and a portion in_1 of its input data, returns a specialized program P_S that is equivalent to P in the sense that when the remaining portion in_2 of the input of P is given, then $P_S(\mathit{in}_2) = P(\mathit{in}_1, \mathit{in}_2)$ [65, 93, 101].

In this thesis we follow an approach which can be regarded as a generalization of the one proposed in [119]. Given an incorrectness triple of the form $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$, where: (i) the imperative program `Prog` is written in a programming language L, and (ii) the formulas φ_{init} and φ_{error} are specified in a logic M, we generate the verification conditions for checking whether or not $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$ holds by: (1) writing the interpreter `I` encoding the semantics of L and M in CLP, and (2) specializing the interpreter `I` with respect to the given incorrectness triple.

The result of this transformation step, called Verification Conditions Generation, is an equivalent CLP program, say `V`, with respect to the properties of interest where the clauses of the interpreter `I` no longer occur. Since any reference to the statements of the original imperative program `Prog` is *compiled away*, the set of clauses `V` represents a set of verification conditions for `Prog`, and we say that they are *satisfiable* if and only if `incorrect` $\notin M($`V`$)$ (or equivalently `V` $\not\models$ `incorrect`).

Thus, the satisfiability of the verification conditions V guarantees that Prog is correct with respect to φ_{init} and φ_{error}.

This chapter is organized as follows. In Section 3.1 we present the syntax of our imperative language and the CLP encoding of imperative programs produced by the CLP Translation step. In Section 3.2 we define the CLP program encoding the semantics of the imperative language. In Section 3.3 we present the proof rules for proving partial correctness of imperative programs. In Section 3.4 we establish the soundness of the encodings presented in Sections 3.1, 3.2 and 3.3. Finally, in Section 3.5, we present the specialization strategy to perform the Verification Conditions Generation step.

3.1 Encoding Imperative Programs into CLP

We consider an imperative programming language, subset of the C language [94], whose abstract syntax is shown in Figure 2. We assume that: (i) every label occurs in every program at most once, (ii) every variable occurrence is either a global occurrence or a local occurrence with respect to any given function definition, (iii) in every program one may statically determine whether any given variable occurrence is either global or local. Note that there are no blocks, and thus no nested levels of locality. We also assume that: (i) there are no definitions of recursive functions, (ii) there are no side effects when evaluating expressions and functions, and (iii) there are neither structures, nor pointers.

A program Prog whose initial declarations are of the form: *type* $z_1; \ldots ;$ *type* z_r, is said to act on the global variables $z_1, \ldots, z_r$. Without loss of generality, we also assume that the last command of every program is $\ell_{\mathtt{h}}$: halt which, when executed, causes program termination (it essentially encodes the return of the main function of C programs).

In order to simplify the analysis we use the C Intermediate Language [115] tool to compile the program into a simplified subset of the C language, and we use the CIL API to define a visitor that realizes the CLP Translation step of the verification framework depicted in Figure 1. In particular, we use the CIL front-end to translate any C program into a simpler program Prog which consists only of labelled commands (for reasons of brevity, we will feel free to say ‘command’, instead of ‘labelled command’). For instance, the while command is reduced to a suitable sequences of if-else and goto commands. This translation makes the task of generating verification conditions for the program easier. In particular, given a C program Prog, our CIL visitor produces a set of facts defining the predicate at(Lab,Cmd) which encodes commands, where Lab is the

$$
\begin{array}{llll}
x, y, \ldots & \in & \mathit{Vars} & \text{(variable identifiers)} \\
f, g, \ldots & \in & \mathit{Functs} & \text{(function identifiers)} \\
\ell, \ell_1, \ldots & \in & \mathit{Labs} & \text{(labels)} \\
\mathit{const} & \in & \mathbb{Z} & \text{(integer constants, character constants, \ldots)} \\
\mathit{type} & \in & \mathit{Types} & (\texttt{int}, \texttt{char}, \ldots) \\
\mathit{uop}, \mathit{bop} & \in & \mathit{Ops} & \text{(unary and binary operators: } +, -, \leq, \ldots) \\
\langle \text{prog} \rangle & ::= & \langle \text{decl} \rangle^* \ \langle \text{fundef} \rangle^* \ \langle \text{labcmd} \rangle^+;\ \ell_{\mathtt{h}} \texttt{: halt} & \text{(programs)} \\
\langle \text{decl} \rangle & ::= & \mathit{type}\ x & \text{(declarations)} \\
\langle \text{fundef} \rangle & ::= & \mathit{type}\ f\ \texttt{(}\langle \text{decl} \rangle^*\texttt{)}\ \texttt{\{}\,\langle \text{decl} \rangle^*\ \langle \text{labcmd} \rangle^+\,\texttt{\}} & \text{(function definitions)} \\
\langle \text{labcmd} \rangle & ::= & \ell\texttt{:}\ \langle \text{cmd} \rangle & \text{(labelled commands)} \\
\langle \text{cmd} \rangle & ::= & x = \langle \text{expr} \rangle & \\
 & | & x = f\ \texttt{(}\langle \text{expr} \rangle^*\texttt{)} & \\
 & | & \texttt{return}\ \langle \text{expr} \rangle & \\
 & | & \texttt{goto}\ \ell & \\
 & | & \texttt{if (}\langle \mathit{expr} \rangle\texttt{)}\ \ell_1\ \texttt{else}\ \ell_2 & \\
\langle \text{expr} \rangle & ::= & \mathit{const}\ \mid\ x\ \mid\ \mathit{uop}\ \langle \text{expr} \rangle\ \mid\ \langle \text{expr} \rangle\ \mathit{bop}\ \langle \text{expr} \rangle & \text{(expressions)}
\end{array}
$$

Figure 2: Abstract syntax of the imperative language. The superscripts $^+$ and * denote non-empty and possibly empty finite sequences, respectively.

label associated with the command `Cmd`.

We show how this encoding can be done through the following example.

Example 2 (Encoding of an imperative program). Let us consider the following program `sum` acting on the global variables x, y, and n:

```
int x, y, n;
while (x<n) {
  x=x+1;
  y=x+y;
}
```

Listing 3.1: Program `increment`

The `while` command is reduced into the following sequence of labelled commands by the CIL visitor:

1. ℓ_0 : `if (x<n)` ℓ_0 `else` $\ell_{\mathtt{h}}$`;`
2. ℓ_1 : `x=x+1;`
3. ℓ_2 : `y=x+y;`
4. ℓ_3 : `goto` ℓ_0`;`
5. $\ell_{\mathtt{h}}$: `halt;`

Then, this sequence of statements is translated into the following set of facts:

1. `at(0,ite(less(id(x),id(n)),1,h)).`
2. `at(1,asgn(id(x),expr(plus(id(x),int(1))),2)).`
3. `at(2,asgn(id(y),expr(plus(id(x),id(y))),3)).`
4. `at(3,goto(0)).`
5. `at(h,halt).`

The facts 1–5 encode the labelled commands at lines 1–5. Note that the definition of `at` also includes information about the control flow of the program. For instance, the third argument of the fact at line 2 represents the label of the command to be executed after the assignment is performed, that is, the command at line 3 .

3.2 Encoding Operational Semantics into CLP

In this section we present the CLP clauses which encode the semantics of the imperative language L shown in Figure 2.

Let us first introduce the following data structures.

(i) A *global environment* δ : *Vars* $\rightarrow \mathbb{Z}$ is a function that maps global variables to their values.

(ii) A *local environment* σ: *Vars* $\rightarrow \mathbb{Z}$ is a function that maps function parameters and local variables to their values.

(iii) An *activation frame* is a triple of the form $\langle \ell, y, \sigma \rangle$, where: (1) ℓ is the label where to jump after returning from a function call, (2) y is the variable that stores the value returned by a function call, and (3) σ is the local environment to be initialized when making a function call.

(iv) A *configuration* is a pair of the form $\langle\!\langle c, e \rangle\!\rangle$, where c is a labelled command e is an *environment* of the form $\langle \delta, \tau \rangle$ consisting of a global environment δ and a list τ of activation frames. We operate on the list τ of activation frames by the usual *head* (*hd*) and *tail* (*tl*) functions and the right-associative list constructor *cons* (:). The empty list is denoted by $[\,]$. Given a function f, a variable identifier x, and an integer v, the term *update*(f, x, v) denotes the function f' that is equal to f, except that $f'(x) = v$.

For any program `Prog` (see Figure 2), for any label ℓ, (i) *at*(ℓ) denotes the command in `Prog` with label ℓ, and (ii) *nextlab*(ℓ) denotes the label of the command in `Prog` that is written *immediately after* the command with label ℓ. Given a function identifier f, *firstlab*(f) denotes the label of the first command of the definition of the function f in `Prog`. For any expression e, any global environment δ, and any local environment σ, $[\![e]\!]\,\delta\,\sigma$ is the integer value of e. For instance, if x is a global variable and $\delta(x)=5$, then $[\![x+1]\!]\,\delta\,\sigma = 6$.

The semantics of our language is defined by a *transition relation*, denoted $\Longrightarrow$, between configurations. That relation, denoted $\Longrightarrow$, is defined by the following rules $R1$–$R5$.

$(R1)$. *Assignment.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma\rangle$ and v be the integer $[\![e]\!]\,\delta\,\sigma$.

If x is a global variable:

$\langle\!\langle \ell\!:\!x\!=\!e,\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(nextlab(\ell)),\ \langle update(\delta, x, v),\ \tau\rangle\rangle\!\rangle$

If x is a local variable:

$\langle\!\langle \ell\!:\!x\!=\!e,\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(nextlab(\ell)),\ \langle\delta,\ \langle \ell', y, update(\sigma, x, v)\rangle : tl(\tau)\rangle\rangle\!\rangle$

Informally, an assignment updates either the global environment δ or the local environment σ of the topmost activation frame $\langle \ell', y, \sigma\rangle$.

$(R2)$. *Conditional.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma\rangle$.

If $[\![e]\!]\,\delta\,\sigma = true$:

$\langle\!\langle \ell\!:\ \texttt{if (}e\texttt{)}\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(\ell_1),\ \langle\delta,\ \tau\rangle\rangle\!\rangle$

If $[\![e]\!]\,\delta\,\sigma = false$:

$\langle\!\langle \ell\!:\ \texttt{if (}e\texttt{)}\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(\ell_2),\ \langle\delta,\ \tau\rangle\rangle\!\rangle$

$(R3)$. *Jump.*

$\langle\!\langle \ell\!:\ \texttt{goto}\ \ell',\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(\ell'),\ \langle\delta,\ \tau\rangle\rangle\!\rangle$

$(R4)$. *Function call.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma\rangle$. Let $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_h\}$ be the set of the formal parameters and the set of the local variables, respectively, of the definition of the function f.

$\langle\!\langle \ell\!:\!x\!=\!f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_k\texttt{)},\ \langle\delta,\ \tau\rangle\rangle\!\rangle \ \Longrightarrow\ \langle\!\langle at(firstlab(f)),\ \langle\delta,\ \langle nextlab(\ell), x, \overline{\sigma}\rangle : \tau\rangle\rangle\!\rangle$

where $\overline{\sigma}$ is a local environment of the form: $\{\langle x_1, [\![e_1]\!]\,\delta\,\sigma\rangle, \ldots, \langle x_k, [\![e_k]\!]\,\delta\,\sigma\rangle, \langle y_1, n_1\rangle, \ldots, \langle y_h, n_h\rangle\}$, for some values $n_1, \ldots, n_h$ in $\mathbb{Z}$ (indeed, when the local variables $y_1, \ldots, y_h$ are declared, they are not initialized). Note that since the values of the n_i's are left unspecified, this transition is nondeterministic.

Informally, a function call creates a new activation frame with the label where to jump after returning from the call, the variable where to store the returned value, and the new local environment.

$(R5)$. *Return.* Let τ be $\langle \ell', y, \sigma\rangle : \langle \ell'', z, \sigma'\rangle : \tau''$ and v be the integer $[\![e]\!]\,\delta\,\sigma$.

If y is a global variable:

$$\langle\!\langle \ell\!:\!\texttt{return}\ e,\ \langle \delta,\ \tau\rangle\rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \langle update(\delta, y, v),\ tl(\tau)\rangle\rangle\!\rangle$$

If y is a local variable:

$$\langle\!\langle \ell\!:\!\texttt{return}\ e,\ \langle \delta,\ \tau\rangle\rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \langle \delta,\ \langle \ell'', z, update(\sigma', y, v)\rangle\!:\!\tau''\rangle\rangle\!\rangle$$

Informally, a `return` command first evaluates the expression e and computes the value v to be returned, then erases the topmost activation frame $\langle \ell', y, \sigma\rangle$, and then updates either the global environment δ or the local environment σ' of the new topmost activation frame $\langle \ell'', z, \sigma'\rangle$.

Obviously, no rule is given for the command `halt`, because no new configuration is generated when `halt` is executed.

The CLP interpreter for the imperative language is given by the following clauses for the binary predicate `tr` that relates old configurations to new configurations and defines the transition relation $\implies$. A configuration of the form $\langle\!\langle c, \langle \delta, \tau\rangle\rangle\!\rangle$, where c is a command and $\langle \delta, \tau\rangle$ is an environment, is encoded by using the term `cf(c,e)`, where `c` and `e` are terms encoding c and $\langle \delta, \tau\rangle$, respectively. We have the following clauses encoding rules $R1$–$R5$: ($R1$) assignments (clause 6), ($R2$) conditionals (clauses 7 and 8), ($R3$) jumps (clause 9), and ($R4$) function calls (clause 10), and ($R5$) function returns (clause 11), defined as follows.

```
6. tr(cf(cmd(L,asgn(Id,Ae,Lp)),E), cf(cmd(Lp,C),Ep)) :-
      aeval(Ae,E,Val), update(Id,Val,E,Ep), at(Lp,C).
```

where: (i) the term `asgn(Id,Ae,Lp)` encodes the assignment of the value of the expression `Ae` to a variable `Id`, and (ii) `E` is a pair of lists of the form `(Glb,Afl)` encoding the global environment δ and the list of activation frames τ, respectively. The predicate `aeval(Ae,E,Val)` computes the value `Val` of the expression `Ae` in the environment `E`. The predicate `update(Id,Val,E,Ep)` updates the environment `E` by binding the variable `Id` to the value `Val`, thereby constructing a new environment `Ep`. In particular, it updates either the list `Glb`, or the list `Loc` encoding the local environment of the topmost activation frame `[Ret,RId,Loc]` in the list `Afl`. The information about the scope of a program variable is encoded in `Id`.

```
7. tr(cf(cmd(L,ite(Be,Lt,Le)),E), cf(cmd(Lt,C),E)) :-
      beval(Be,E), at(Lt,C).
8. tr(cf(cmd(L,ite(Be,Lt,Le)),E), cf(cmd(Le,C),E)) :-
      beval(not(Be),E), at(Le,C).
```

where the term `ite(Be,Lt,Le)` encodes the boolean expression `Be` and the labels `Lp` and `Le` where to jump according to the evaluation `beval(Be,E)` of `Be` in the environment `E`. The predicate `beval(Be,E)` holds if `Be` is true in the environment `E`.

9. `tr(cf(cmd(L,goto(Lp)),E), cf(cmd(Lp,C),E)) :- at(Lp,C).`

where `Lp` is the label of the next command `cmd(Lp,C)` to be executed.

10.
```
tr(cf(cmd(L,call(F)),E),cf(cmd(Lp,C),Ep)) :-
      prologue(F,E,Lp,Ep), at(Lp,C).
```

where `F` is a list of the form `[Ael,RId,Lp,Ret]`, defined as follows: (i) `Ael` is the list of the actual parameters, (ii) `RId` is the variable where to store the returned value, (iii) `Lp` is the label of the first command `C` in the definition of the function `F`, and (iv) `Ret` is the label where to jump after returning from the function call. The predicate `prologue(F,E,Lp,Ep)`: (1) builds the new local environment `Loc`, where the body of the function should be executed, by evaluating the list of the actual parameters `Ael` in the environment `E`, and (2) adds a new activation frame of the form `[Ret,RId,Loc]` to `E` thereby producing a new environment `Ep`.

11.
```
tr(cf(cmd(L,ret(Ae)),E),cf(cmd(Lp,C),Ep)) :-
      epilogue(Ae,E,Lp,Ep), at(Lp,C).
```

where `ret(Ae)` encodes the expression `Ae` returned from a function call. The predicate `epilogue(Ae,E,Lp,Ep)`: (1) computes the value `Val` of the expression `Ae` in the environment `E`, (2) updates the environment `E` by binding the variable `RId` (that is, the variable where to store the value returned by the called funtion) to the value `Val`, and (3) erases the topmost activation frame `[Lp,RId,Loc]` from `E`, thereby producing a new environment `Ep`.

Note that the CLP clauses 6–11 are clauses without constraints in their bodies. However, constraints are used in the definitions of the predicates `aeval` and `beval`.

3.3 Encoding Partial Correctness into CLP

In this section we introduce the clauses which encode the semantics of the logic M, that is, the reachability relation which allows us to prove the partial correctness of imperative programs.

The problem of verifying the correctness of a programs `Prog` is the problem of checking whether or not, starting from an initial configuration, the execution of `Prog` leads to a so-called error configuration. This problem is formalized by defining an incorrectness triple of the form:

$$\{\!\{\varphi_{init}(z_1,\ldots,z_r)\}\!\}\ \texttt{Prog}\ \{\!\{\varphi_{error}(z_1,\ldots,z_r)\}\!\}$$

where:

(i) `Prog` is a program acting on the global variables $z_1,\ldots,z_r$,

(ii) $\varphi_{init}(z_1,\ldots,z_r)$ is a disjunction of constraints that characterizes the values of the global variables in the initial configurations, and

(iii) $\varphi_{error}(z_1,\ldots,z_r)$ is a disjunction of constraints that characterizes the values of the global variables in the error configurations.

Note that our notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}\ prog\ \{\neg\varphi_{error}\}$.

We say that a program `Prog` is *incorrect* with respect to a set of initial configurations satisfying $\varphi_{init}(z_1,\ldots,z_r)$ and a set of error configurations satisfying $\varphi_{error}(z_1,\ldots,z_r)$ or simply, `Prog` is *incorrect* with respect to φ_{init} and φ_{error}, if there exist two global environments δ_{init} and δ_{halt} such that the following conjunction holds:

$$\begin{array}{ll} & \varphi_{init}(\delta_{init}(z_1),\ldots,\delta_{init}(z_r)) \\ \wedge & \langle\!\langle c_{init},\langle\delta_{init},[\,]\rangle\rangle\!\rangle \Longrightarrow^{*} \langle\!\langle c_{halt},\langle\delta_{halt},[\,]\rangle\rangle\!\rangle \\ \wedge & \varphi_{error}(\delta_{halt}(z_1),\ldots,\delta_{halt}(z_r)) \end{array} \qquad (\dagger)$$

where: c_{init} is the first command of `Prog` and, as already mentioned, c_{halt} is the last command of `Prog`. As usual, $\Longrightarrow^{*}$ denotes the reflexive, transitive closure of $\Longrightarrow$. The definition of incorrectness (†) can be easily translated into CLP as follows:

```
12. incorrect :- initConf(X), reach(X).
13. reach(X) :- tr(X,X1), reach(X1).
14. reach(X) :- errorConf(X).
15. initConf(cf(cmd(0,C),E)) :- at(0,C), phiInit(E).
16. errorConf(cf(cmd(h,halt),E)) :- phiError(E).
```

where: (i) the predicate `tr` represents the transition relation $\Longrightarrow$, (ii) the predicate `reach` represents the reachability of the error configurations, (iii) the predicate `initConf` represents the initial configurations, (iv) the predicate `phiInit` represents the initial property φ_{init}, (v) the predicate `errorConf` represents the error configurations, and (vi) the predicate `phiError` represents the error property φ_{error}.

Example 3 (Enconding of an incorrectness triple). Let us consider the incorrectness triple $\{\!\{\varphi_{init}(x,y,n)\}\!\}$ `sum` $\{\!\{\varphi_{error}(x,y,n)\}\!\}$, where: (i) $\varphi_{init}(x,y,n)$

is $x=0 \wedge y=0$, (ii) $\varphi_{error}(x,y,n)$ is $x>y$, and (iii) `sum` is the program defined in Example 2.

The formulas $\varphi_{init}(x,y,n)$ and $\varphi_{error}(x,y,n)$ are encoded as follows:

```
17. phiInit(([[int(x),X],[int(y),Y],[int(n),N]],[])) :- X=0, Y=0.
18. phiError(([[int(x),X],[int(y),Y],[int(n),N]],[])) :- X>Y.
```

In clauses 17 and 18 the pair `([[int(x),X],[int(y),Y],[int(n),N]],[])` encodes the program environment. The first component, that is the global environment, is the list `[[int(x),X],[int(y),Y],[int(n),N]]` that provides the bindings for the global variables x, y, and n, respectively. The second component, that is, the list of activation frames, is the empty list `[]`. $\square$

3.4 Soundness of the CLP Encoding

In this section we establish the soundness of the CLP encodings presented in the previous sections.

Let us first introduce the definition of *CLP Encoding*.

Definition 6 (CLP Encoding). Let us consider an incorrectness triple of the form $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$. Let `T` be the CLP program consisting of the clauses that defines the predicates `phiInit`, `at`, and `phiError` encoding the initial property φ_{init}, the imperative program `Prog`, and the error property φ_{error}, respectively. Let `I` the CLP program consisting of the clauses 6–16 encoding the interpreter for the correctness problem. The CLP program, say it `P`, which is the union of `T` and `I`, is called the *CLP Encoding* of the correctness problem for $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$.

The following result establishes that the CLP Encoding is sound.

Theorem 2 (Soundness of the CLP Encoding). Let `P` be the CLP Encoding of the correctness problem for $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$. The program `Prog` is correct with respect to φ_{init} and φ_{error} if and only if `incorrect` $\notin M($`P`$)$.

Proof. In the CLP program `P`, the predicate `tr` encodes the transition relation $\Longrightarrow$ associated with the given imperative program `Prog`, that is, `P` $\models$ `tr(cf1,cf2)` if and only if $cf_1 \Longrightarrow cf_2$, where `cf1` and `cf2` are terms encoding the configurations cf_1 and cf_2, respectively. The predicates `initConf` and `errorConf` encode the initial and error configurations, respectively, that is, the following Properties (A) and (B) hold.

Property (A): `P` $\models$ `initConf(init-cf)` iff `init-cf` is the term encoding a configuration of the form $\langle\!\langle \ell_0 : c_0, \langle \delta_{init}, [\,]\rangle\rangle\!\rangle$ such that $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds, and
Property (B): `P` $\models$ `errorConf(error-cf)` iff `error-cf` is the term encoding a configuration of the form $\langle\!\langle \ell_h : \mathtt{halt}, \langle \delta_{halt}, [\,]\rangle\rangle\!\rangle$ such that $\varphi_{error}(\delta_{halt}(z_1), \ldots, \delta_{halt}(z_r))$ holds.

By clauses 13 and 14 of the CLP program `P` and Property (B), for any configuration *cf* encoded by the term `cf`, we have that `P` $\models$ `reach(cf)` iff there exists a configuration cf_h of the form $\langle\!\langle \ell_h : \mathtt{halt}, \langle \delta_{halt}, [\,]\rangle\rangle\!\rangle$ such that $\varphi_{error}(\delta_{halt}(z_1), \ldots, \delta_{halt}(z_r))$ holds and $cf \Longrightarrow^* cf_h$.

Now, by clause 12 of the CLP program `P` and Property (A), we get that `P` $\models$ `incorrect` iff there exist configurations cf_0 and cf_h such that the following hold:
(i) cf_0 is of the form $\langle\!\langle \ell_0 : c_0, \langle \delta_{init}, [\,]\rangle\rangle\!\rangle$,
(ii) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$,
(iii) $cf_0 \Longrightarrow^* cf_h$,
(iv) cf_h is of the form $\langle\!\langle \ell_h : \mathtt{halt}, \langle \delta_{halt}, [\,]\rangle\rangle\!\rangle$, and
(v) $\varphi_{error}(\delta_{halt}(z_1), \ldots, \delta_{halt}(z_r))$.

Thus, by the definition of incorrectness, `P` $\models$ `incorrect` iff `Prog` is incorrect with respect to the properties φ_{init} and φ_{error}. The thesis follows from the fact that `P` $\models$ `incorrect` iff `incorrect` $\in M(\mathtt{P})$ [84]. □

3.5 The Specialization Strategy

In this section we present a transformation strategy, called $\mathit{Specialize}_{vcg}$, which realizes the Verification Conditions Generation step of the verification framework. In particular, $\mathit{Specialize}_{vcg}$ unfolds away the relations on which `incorrect` depends and introduces new predicate definitions corresponding to (a subset of) the 'program points' of the original imperative program. The transformation strategy $\mathit{Specialize}_{vcg}$ specializes the CLP Interpreter `I`, consisting of clauses 6–16, with respect to the CLP encoding of the incorrectness triple `T`, consisting of the clauses defining the predicates `at`, `phiInit`, and `phiError`.

The result of this first transformation step is a new CLP program `V`, such that `incorrect` $\in M(\mathtt{P})$ iff `incorrect` $\in M(\mathtt{V})$.

The transformation strategy $\mathit{Specialize}_{vcg}$, shown in Figure 3, makes use of the following transformation rules: *Unfolding*, *Clause removal*, *Definition introduction*, and *Folding*.

In order to perform the UNFOLDING phase, we need to annotate the atoms occurring in bodies of clauses as either *unfoldable* or *not unfoldable*. This annotation, which is a critical part of a transformation strategy, ensures that any sequence of clauses constructed by unfolding w.r.t. unfoldable atoms is finite. We will see a possible choice for this annotation in Example 4. We refer to [101] for a survey of techniques for controlling unfolding that guarantee this finiteness property.

In order to perform the DEFINITION INTRODUCTION phase, we make use of a set *Defs* of *definitions* arranged as a *tree* whose root is the initial input clause `incorrect:- initConf(X), reach(X)` considered at the beginning of the specialization process. Each new definition D introduced during specialization determines a new node of a tree which is placed as a *child* of the definition C, and represented by $child(D, C)$, if it has been introduced to fold a clause obtained by unfolding C. The root is represented by $child(D, \mathtt{T})$. We define the *ancestor* relation as the reflexive, transitive closure of the child relation.

3.5.1 Termination and Soundness of the Specialization Strategy

The following theorem establishes the termination and soundness of the *Specialize$_{vcg}$* strategy.

Theorem 3 (Termination and Soundness of the *Specialize$_{vcg}$* strategy). (i) The *Specialize$_{vcg}$* strategy terminates. (ii) Let program `V` be the output of the *Specialize$_{vcg}$* strategy applied on the input program `P`. Then `incorrect` $\in M(\mathtt{P})$ iff `incorrect` $\in M(\mathtt{V})$.

Proof. (i) In order to prove the termination of the *Specialize$_{vcg}$* procedure we assume that the *unfoldable* annotations and *not unfoldable* annotations guarantee the termination of the UNFOLDING while-loop ($\alpha1$). Since the CLAUSE REMOVAL while-loop ($\alpha2$), the DEFINITION INTRODUCTION while-loop ($\alpha3$), and the FOLDING while-loop (β) clearly terminate, we are left with showing that the first, outermost while-loop (α) terminates, that is, a finite number of new predicate definitions is added to *InCls* by DEFINITION INTRODUCTION. This finiteness is guaranteed by the following facts:
(1) all new predicate definitions are of the form `newk(Y) :- reach(cf(cmd,E))` where `reach(cf(cmd,E))` is the atom with fresh new variables occurring in the body of the clause to be folded (this construction ensures that every other clause whose atom occurring in the body is a variant of `reach(cf(cmd,E))` can be folded using such a definition), and (2) only a finite set of configurations is

Input: Program P.
Output: Program V such that incorrect $\in M$(P) iff incorrect $\in M$(V).

```
INITIALIZATION:
V := $\emptyset$;
$InCls$ := { incorrect :- initConf(X), reach(X) };
$Defs$ := $\emptyset$;
  ($\alpha$) while in $InCls$ there is a clause $C$ that is not a constrained fact do
    UNFOLDING:
      $SpC := Unf(C, A)$, where $A$ is the leftmost atom in the body of $C$;
      ($\alpha 1$) while in $SpC$ there is a clause $D$ whose body contains an
                 occurrence of an unfoldable atom $A$ do
             $SpC := (SpC - \{D\}) \cup Unf(D, A)$;
           end-while;
    CLAUSE REMOVAL:
      ($\alpha 2$) while in $SpC$ there are two distinct clauses $D_1$ and $D_2$ such that
                 $D_1$ subsumes $D_2$ do
             $SpC := SpC - \{D_2\}$;
           end-while;
    DEFINITION INTRODUCTION:
      ($\alpha 3$) while in $SpC$ there is a clause $D$ of the form
                 H(X) :- c(X,Y), reach(cf(cmd,E)), where: c(X,Y) is a
                 constraint and cf(cmd,E) is a configuration with Y $\subseteq$ $vars$(E),
                 such that it cannot be folded using a definition in $Defs$ do
             $Defs$ := $Defs$ $\cup$ { $child$(newk(Y) :- reach(cf(cmd,E)), $C$)};
             $InCls$ := $InCls$ $\cup$ { newk(Y) :- reach(cf(cmd,E))};
           end-while;
    $InCls := InCls - \{C\}$;
    V := V $\cup$ $SpC$;
  end-while;
FOLDING:
  ($\beta$) while in V there is a clause $E$ that can be folded
          by a clause $D$ in $Defs$ do
       V := (V $-$ {$E$}) $\cup$ {$F$}, where $F$ is derived by folding $E$ using $D$;
     end-while;
Remove from V all clauses for predicates on which incorrect does not depend.
```

Figure 3: The $Specialize_{vcg}$ Procedure.

generated during the specialization process. All configurations are of the form `cf(cmd,E)`, where: (2.1) the term `cmd` ranges over the set of commands belonging to the imperative program, and (2.2) the term `E` ranges over the set of terms encoding environments. Since the set of commands in a program is finite, then the set at point (2.1) is finite. An environment `E` is encoded as a pair of list: a list of global variables and a list of activation frames (each of which contains a local environment). Since the number of program variables is finite, then the global and the local environments are finite lists. Moreover, since we do not consider recursive functions, then the list of activation frames is finite. Hence, the list at point (2.2) is finite. (3) no two new predicate definitions that are equal modulo the head predicate name are introduced by DEFINITION INTRODUCTION (indeed, DEFINITION INTRODUCTION introduces a new predicate definition only if the definitions already present in *Defs* cannot be used to fold clause E).
(ii) in order to prove the soundness of $\mathit{Specialize}_{vcg}$ we need to ensure that $\mathit{Specialize}_{vcg}$ enforces all the applicability conditions for the unfolding and folding rules presented in Chapter 2. The $\mathit{Specialize}_{vcg}$ procedure constructs a transformation sequence $P_0, \ldots, P_n$, such that:
(1) P_0 is `P`, and
(2) P_n is $(\mathtt{P} - \{\mathtt{incorrect:\text{-}\ initConf(X),\ reach(X)}\}) \cup P'$, where P' is the value of `V` at the exit of the FOLDING while-loop (β).

The hypothesis of the Theorem 1 (Correctness of the Transformation Rules) is fulfilled, as all clauses in *Defs* are unfolded. Thus, `incorrect` $\in M(\mathtt{P})$ iff `incorrect` $\in M(P_n)$. The thesis follows from the fact that, by deleting from P_n the clauses defining predicates on which `incorrect` does not depend, we get a final program `V` such that `incorrect` $\in M(\mathtt{V})$ iff `incorrect` $\in M(P_n)$. □

Example 4 (Generating Verification Conditions). Let us consider again the incorrectness triple $\{\!\{x = 0 \land y = 0\}\!\}$ `sum` $\{\!\{x > y\}\!\}$. In order to generate the verification conditions for `sum` we apply the $\mathit{Specialize}_{vcg}$ strategy.

In order to guarantee the termination of the *Unf* procedure, an atom `A` in the body of a clause *Cl* is *unfoldable* iff one of the following conditions holds:
(i) the predicate of `A` is different from `reach`,
(ii) `A` is of the form `reach(cf(cmd(L,C),E))` and `C` is either `ret(Ae)`, or `asgn(Id,expr(Ae),L1)`, or `halt`,
(iii) `A` is of the form `reach(cf(cmd(L,goto(L1)),E))` and *Cl* has *not* been derived (in one or more steps) by unfolding a clause with respect to an atom of the form `reach(cf(cmd(L,goto(L1)),E))`.
Note that a `reach` atom containing a command of the form `asgn(x,call(...))`

is assumed to be not unfoldable. Condition (iii) allows unfolding with respect to a `reach` atom containing a `goto` command, but prevents infinite unfolding. (Recall that we have assumed that the imperative program `Prog` does not contain definitions of recursive functions.) Finally, note that a `reach` atom containing an `ite` command is not unfoldable, and hence a potential exponential blowup due to the unfolding of conditionals is avoided. Indeed, it can easily be shown that the size of the output `V` of *Specialize*$_{vcg}$ is linear with respect to the size of `P` (and thus, also with respect to the size of the imperative program `Prog` of the triple encoded by `T`).

In order to perform folding, a clause C is folded using either (i) the clause introduced during the Definition introduction, (ii) the clause, say D, from which C has been derived, or (iii) the most recent ancestor of D.

We apply the *Specialize*$_{vcg}$ strategy as we now show.

The Initialization procedure performs the assignments: `V` $:= \emptyset$, *InCls* $:=$ $\{$`incorrect :- initConf(X), reach(X)`$\}$ (that is, clause 12), and *Defs*$:=\emptyset$.

Since *InCls* $\neq \emptyset$, *Specialize*$_{vcg}$ enters the while-loop (α). First, the Unfolding loop ($\alpha1$) executes the assignment *SpC*$:=$*Unf*(12, `initConf(X)`), which unfolds clause 12 w.r.t. the atom `initConf(X)`. We get:

```
19. incorrect :- X=0, Y=0,
      reach(cf(cmd(0,ite(less(int(x),int(n)),l1,h)),
      [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

The `reach` atom in clause 19 is not unfoldable because it contains an `ite` command, hence, the while-loop ($\alpha1$) terminates. No clause can be removed by Clause Removal loop ($\alpha2$), and therefore, the while-loop (α) continues by executing the Definition Introduction loop ($\alpha3$). In order to fold clause 19 the Definition-Introduction introduces the following clause:

```
20. new1(X,Y,N) :- reach(cf(cmd(0,ite(less(int(x),int(n)),l1,h)),
      [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

and performs the assignments: *Defs* $:=$ *Defs* $\cup$ $\{$*child*$(20,$ `T`$)\}$ $= \{20\}$, and *InCls* $:=$ *InCls* $\cup$ $\{20\} = \{12, 20\}$.

The first execution of the while-loop (α) terminates by performing the following assignments: *InCls* $:=$ *InCls* $-$ $\{12\} = \{20\}$, and `V` $:=$ `V` $\cup$ *SpC* $= \{19\}$.

Since *InCls* $\neq \emptyset$, we perform a second iteration of the while-loop (α). The Unfolding procedure executes *SpC* $:=$ *Unf*(20, `reach(cf(cmd(0,ite(...)))`). We get two clauses:

```
21. new1(X,Y,N) :- tr(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
      [[int(x),X],[int(y),Y],[int(n),N]],[]),Z), reach(Z).
```

22. new1(X,Y,N) :- errorConf(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
    [[int(x),X],[int(y),Y],[int(n),N]],[])).

The tr and errorConf atoms in the bodies of clauses 21 and 22, respectively, are unfoldable. Thus, the while-loop ($\alpha 1$) perform some more unfolding steps and from clause 21, after a few steps that can be viewed as mimicking the symbolic evaluation of the conditional command, we get the following two clauses:

23. new1(X,Y,N) :- X<N,
    reach(cf(cmd(1,asgn(int(x),expr(plus(int(x),int(1))))),
    [[int(x),X],[int(y),Y],[int(n),N]],[])).

24. new1(X,Y,N) :- X≥N,
    reach(cf(cmd(h,halt),[[int(x),X],[int(y),Y],[int(n),N]],[])).

Clauses 23 and 24 represent the branches of the conditional command of the imperative program sum. Indeed, the test on the condition less(int(x),int(n)) in the command of clause 21 generates the two constraints X<N and X≥N.

Then, we delete clause 22 because by unfolding it we derive the empty set of clauses (indeed, the term cmd(0,...) does not unify with the term cmd(h,...)).

The reach atom in the body of clause 23 is unfoldable, because it contains a command of the form asgn(int(x), expr(...)). From clause 23, after two unfolding steps executed by the while-loop ($\alpha 1$), we get:

25. new1(X,Y,N) :- X<N,
    tr(cf(cmd(1,asgn(int(x),expr(plus(int(x),int(1))))),
    [[int(x),X],[int(y),Y],[int(n),N]],[]),Z)),reach(Z).

Then, by unfolding clause 25 with respect to tr atom in its body, we get:

26. new1(X,Y,N) :- X<N, X1=X+1,
    reach(cf(cmd(2,asgn(int(y),expr(plus(int(x),int(y))))),
    [[int(x),X1],[int(y),Y],[int(n),N]],[])).

The reach atom in the body of clause 26 is unfoldable. After two unfolding steps, we get:

27. new1(X,Y,N) :- X<N, X1=X+1,
    tr(cf(cmd(1,asgn(int(y),expr(plus(int(x),int(y))))),
    [[int(x),X1],[int(y),Y],[int(n),N]][]),Z)), reach(Z).

By unfolding clause 27 we get:

28. new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, reach(cf(cmd(3,goto(l0)),
    [[int(x),X1],[int(y),Y1],[int(n),N]],[])).

The sequence of clauses 19, 23, 26, and 28, which we have obtained by unfolding, mimics the execution of the sequence of the four commands: (i) ℓ_0: if(x<n) ℓ_1;

`else` $\ell_{\mathtt{h}}$, (ii) ℓ_1: `x=x+1`, (iii) ℓ_2: `y=x+y`, and (iv) ℓ_3: `goto` ℓ_0 (note that in those clauses the atoms `reach(cf(cmd(`i`,C),E))`, for $i=0,1,2,3$). Indeed, in general, by unfolding one can to perform the symbolic execution of the commands of any given program. The conditions that should hold so that a particular command `cmd(`i`,C)` is executed, are given by the constraints in the clause where the atom `reach(cf(cmd(`i`,C),E))` occurs.

The `reach` atom in the body of clause 28 is unfoldable, because clause 28 has not been derived from another clause containing a `goto` command. From clause 28, after some more unfolding steps, we get:

```
29. new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y,
      reach(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
      [[int(x),X1],[int(y),Y1],[int(n),N]],[])).
```

The `reach` atom in the body of clause 29 is not unfoldable, because it contains the `ite` command.

Also the `reach` atom in the body of clause 24, which represents the else branch of the conditional command, is unfoldable. After two unfolding steps, we get:

```
30. new1(X,Y,N) :- X≥N, errorConf(cf(cmd(h,halt),
      [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

Then, by unfolding clause 30 we get:

```
31. new1(X,Y,N) :- X≥N, X>Y.
```

Since in clauses 29 and 31 no unfoldable atoms occur the while-loop (α1) terminates with $SpC := \{29, 31\}$.

The Definition Introduction procedure terminates without introducing any new definition. Indeed, clause 29 can be folded using definition 20. The second execution of the while-loop (α) terminates by performing the assignments: $InCls := InCls - \{20\} = \emptyset$, and $\mathtt{V} := \mathtt{V} \cup SpC = \{19,\ 29,\ 31\}$.

Since $InCls = \emptyset$, the while-loop (α), proceeds by executing the Folding procedure, which concludes the $Specialize_{vcg}$ strategy. By folding clauses {19, 29} using definition 20 we get the following final program `V`:

```
32. incorrect :- X=0, Y=0, new1(X,Y,N).
33. new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
34. new1(X,Y,N) :- X≥N, X>Y.
```

Note that the application of the folding rule on clause 29 using the definition for predicate `new1` has been possible because the execution of the program goes back to the `ite` command to which the definition of `new1` refers. □

3.6 Related Work

The idea of encoding imperative programs into CLP programs for reasoning about their properties was presented in various papers [16, 63, 88, 119, 127], which show that through CLP programs one can express in a simple manner both (i) the symbolic executions of imperative programs, and (ii) the invariants that hold during their executions.

The use of constraint logic program specialization for analyzing imperative programs has also been proposed by [119], where the interpreter of an imperative language is encoded as a CLP program. Then the interpreter is specialized with respect to a specific imperative program to obtain a residual program on which a static analyzer for CLP programs is applied.

The verification method presented in [63] is based on a semantics preserving translation from an imperative language with dynamic data structures and recursive functions into CLP. This translation reduces the verification of the (in)correctness of imperative programs to a problem of constraint satisfiability within standard CLP systems.

CHAPTER 4

Verifying Programs by Specializing Verification Conditions

In this chapter we show how program specialization can be used not only as a preprocessing step to generate verification conditions, but also as a means of analysis on its own, as an alternative to static analysis techniques of CLP programs.

Let us consider an incorrectness triple of the form $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$. According to the method proposed in Chapter 3 we have that the CLP program `V`, obtained at the end of the Verification Conditions Generation step, consists of a set of clauses representing the verification conditions for proving the partial correctness of `Prog` with respect to φ_{init} and φ_{error}. By Theorems 2 and 3, checking the satisfiability of the verification conditions for `Prog` reduces to check whether or not the atom `incorrect` is a consequence of `V`.

Unfortunately, the problem of deciding whether or not `incorrect` is a consequence of `V` is undecidable. Consequently, verification methods based on top-down, bottom-up, and tabled query evaluation strategies may not terminate. In order to cope with this undecidability limitation, and improve the termination of the verification process, we propose a strategy based on program specialization. In particular, instead of applying program analysis techniques to the CLP program `V`, in the Verification Conditions Transformation step we further specialize `V` with respect to the initial property φ_{init}, thereby deriving a new CLP program `S`, which is equivalent to `V` with respect to the property of interest, that is, `incorrect` $\in M($`V`$)$ iff `incorrect` $\in M($`S`$)$. The effect of this further transformation is the modification of the structure of `V` and the explicit addition of new constraints that denote invariants of the computation. Through various experiments we show that by exploiting these invariants, the construction of the least model of the CLP program S, which is realized in the Verification Conditions Analysis step through a bottom-up evaluation procedure, terminates in many interesting cases and, thus, it is possible to verify the correctness of Prog with respect to φ_{init} and φ_{error} by simply inspecting that model.

An essential ingredient of program specialization are the *generalization operators*, which introduce new predicate definitions representing invariants of the program executions. Generalizations are used to enforce the termination of program specialization (recall that program specialization terminates when no new predicate definitions are introduced) and, in this respect, they are similar to the widening operators used in static program analysis [30, 34]. One problem encountered with generalizations is that sometimes they introduce predicate definitions which are too general, thereby making specialization useless. We introduce a new generalization strategy, called *constrained generalization*, whose objective is indeed to avoid the introduction of new predicate definitions that are too general.

The basic idea of constrained generalization is related to the branching behavior of the unfolding steps, as we now indicate. Given a sequence of unfolding steps performed during program specialization, we may consider a symbolic evaluation tree made out of clauses, such that every clause has as children the clauses which are generated by unfolding that clause. Suppose that a clause γ has n children which are generated by unfolding using clauses $\gamma_1, \ldots, \gamma_n$, and suppose that during program specialization we have to generalize clause γ. Then, we would like to perform this generalization by introducing a new predicate definition, say δ, such that by unfolding clause δ, we get again, if possible, n children and these children are due to the same clauses $\gamma_1, \ldots, \gamma_n$.

Since in this generalization the objective of preserving, if possible, the branching structure of the symbolic evaluation tree, is realized by adding extra constraints to the clause obtained after a usual generalization step (using, for instance, the widening operator [30] or the convex-hull operator [34]), we call this generalization *a constrained generalization*. Similar proposals have been presented in [15, 78] and in Section 4.4 we will briefly compare those proposals with ours.

This chapter is organized as follows. In Section 4.1 we outline our software model checking method by developing an example taken from [75]. In Sections 4.2 we describe our strategy for specializing CLP programs, and in Section 4.2.1 we presents some generalization operator and, in particular, our novel constrained generalization technique. In Section 4.3 we report on some experiments we have performed by using a prototype implementation based on the MAP transformation system [108]. We also compare the results we have obtained using the MAP system with the results we have obtained using state-of-the-art software model checking systems such as ARMC [123], HSF(C) [73], and TRACER [85].

4.1 The Verification Method

In this section we outline our method for software model checking which is obtained from the general verification framework (see Figure 1) by providing suitable subsidiary procedures that realize Step 1–Step 4.

The Software Model Checking Method

Input: An incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$ and the CLP program `I` defining the predicate `incorrect`.

Output: The answer *correct* iff `Prog` is correct with respect to φ_{init} and φ_{error}.

```
Step 1: T := $C2CLP(\texttt{Prog}, \varphi_{init}, \varphi_{error})$;   P := T ∪ I;
Step 2: V := $Specialize_{vcg}(\texttt{P})$;
Step 3: S := $Specialize_{prop}(\texttt{V})$;
Step 4: $M(\texttt{S})$ := $BottomUp(\texttt{S})$;
Return the answer correct iff incorrect $\notin M(\texttt{S})$.
```

Figure 4: The Verification Method

The CLP Translation step (Step 1) and the Verification Conditions Generation step (Step 2), of the verification method shown in Figure 4, rely on the *C2CLP* and *Specialize*$_{vcg}$ procedures, respectively. In particular, in order to guarantee the termination of the *Unf* subsidiary procedure of *Specialize*$_{vcg}$ (see Figure 3), an atom `A` is selected for unfolding only if it has not been derived by unfolding a variant of `A` itself.

The verification method shown in Figure 4 avoids the direct evaluation of the clauses in the CLP program `V` and applies symbolic evaluation methods based on program specialization. Indeed, starting from the CLP program `V`, the Verification Conditions Transformation step (Step 3) performs a further specialization, called *the propagation of the constraints*, which consists in specializing `V` with respect to the constraint representing the initial property φ_{init}, with the aim of deriving, if possible, a CLP program `S` whose least model $M(\texttt{S})$ is a finite set of constrained facts. The least model $M(\mathtt{S})$ is computed by using a bottom-up evaluation procedure.

In order to perform the Verification Conditions Transformation step, we propose a specialization strategy, called *Specialize*$_{prop}$ (see Figure 6), which extends *Specialize*$_{vcg}$ with a more powerful DEFINITION INTRODUCTION phase. In particular, the DEFINITION INTRODUCTION of *Specialize*$_{prop}$ makes use of *generalization operators* that are related to some *abstract interpretation* techniques [30] and they play a role similar to that of abstraction in the verification methods described in [26, 36, 1]. However, since it is applied *during* the verification process, and *not before* the verification process, our generalization is often more flexible than abstraction.

By means of an example borrowed from [75], we argue that program specialization can prove program correctness in some cases where the CEGAR method (as implemented in ARMC [123]) does not work. In particular we show that the construction of the least model $M(\mathtt{S})$ terminates and we can prove the correctness of the imperative program `Prog` with respect to φ_{init} and φ_{error} by showing that the atom `incorrect` does not belong to that model.

Example 5 (Proving the partial correctness of an imperative program). Let us consider the following incorrectness triple

$\{\!\{\varphi_{init}(x,y,n)\}\!\}$ `doubleLoop` $\{\!\{\varphi_{error}(x,y,n)\}\!\}$

where: (i) $\varphi_{init}(x,y,n)$ is $x\!=\!0 \wedge y\!=\!0, n \geq 0$ (ii) $\varphi_{error}(x,y,n)$ is $x\!<\!y$, and (iii) `doubleLoop` is the program:

```
while (x<n) {
  x = x+1;
  y = y+1;
}
while (x>0) {
  x = x-1;
  y = y-1;
}
```

Listing 4.1: Program `doubleLoop`

We want to prove that `doubleLoop` is correct with respect to $\varphi_{init}(x,y,n)$ and $\varphi_{error}(x,y,n)$, that is, there is no execution of `doubleLoop` with input values of `x`, `y`, and `n` satisfying $\varphi_{init}(x,y,n)$, such that a configuration satisfying $\varphi_{error}(x,y,n)$ is reached.

As shown in Table 4.1 of Section 4.3, CEGAR fails to prove this property, because an infinite set of counterexamples is generated (see the entry '∞' for Program `doubleLoop` in the ARMC column). Conversely, by applying the specialization-based software model checking method depicted in Figure 4 we will be able to prove that `doubleLoop` is indeed correct. By performing the CLP Translation step (Step 1) and the Verification Conditions Generation step (Step 2) of our framework we get the following CLP clauses encoding the verification conditions for `doubleLoop`.

```
1. incorrect :- a(X,Y,N), new1(X,Y,N).
2. new1(X,Y,N) :- X<N, X1=X+1, Y1=Y+1, new1(X1,Y1,N).
3. new1(X,Y,N) :- X≥1, X≥N, X1=X-1, Y1=Y-1, new2(X1,Y1,N).
4. new1(X,Y,N) :- X≤0, X≥N, b(X,Y,N).
5. new2(X,Y,N) :- X≥1, X1=X-1, Y1=Y-1, new2(X1,Y1,N).
6. new2(X,Y,N) :- X≤0, b(X,Y,N).
```

where:

```
7. a(X,Y,N) :- X=1, Y=1, N≥1.
8. b(X,Y,N) :- X<Y.
```

encode the *specialized* initial and error configurations (note that according to the *Unf* function of the $\mathit{Specialize}_{vcg}$ procedure each loop is unrolled once, and therefore we get the atomic constraint occurring in clause 8).

Unfortunately, it is not possible to check by direct evaluation whether or not the atom `incorrect` is a consequence of the above CLP clauses. Indeed, the evaluation of the query `incorrect` using the standard top-down strategy enters into an infinite loop. Tabled evaluation [35] does not terminate either, as infinitely many tabled atoms are generated. Analogously, bottom-up evaluation is unable to return an answer, because infinitely many facts for `new1` and `new2` should be generated for deriving that `incorrect` is not a consequence of the given clauses.

Then, the Verification Conditions Transformation step (Step 3) specializes the CLP program `V` with respect to the property φ_{init}, thereby deriving the specialized program `S`. During Step 3 the constraints occurring in the definitions of `new1` and `new2` are generalized according to a suitable generalization strategy based both on widening [30, 58, 62] and on the novel constrained generalization strategy.

We proceed by following the specialization pattern described in Figure 3 for the Step 2, but in this specialization, we also show a novel definition introduction approach.

We start off by unfolding clause 1 with respect to `a` and we get:

9. `incorrect:- X=1, Y=1, N≥1, new1(X,Y,N).`

Since no clause in *Defs* can be used to fold clause 9 we introduce the following definition:

10. `new3(X,Y,N) :- X=1, Y=1, N≥1, new1(X,Y,N).`

Each new definition introduced during specialization determines a new node of a tree, called *Defs*, whose root is clause 10, which is the first definition we have introduced. The tree *Defs* of all the definitions introduced during the Verification Conditions Transformation step, can be depicted as in Figure 5.

Then, we unfold clause 10 and we get:

11. `new3(X,Y,N) :- X=1, Y=1, N≥2, X1=2, Y1=2, new1(X1,Y1,N).`
12. `new3(X,Y,N) :- X=1, Y=1, N=1, X1=0, Y1=0, new2(X1,Y1,N).`

Now, we should fold these two clauses. Let us deal with them, one at the time, and let us first consider clause 11. In order to fold clause 11 we consider a definition, called the *candidate definition*, which is of the form:

13. `new4(X,Y,N) :- X=2, Y=2, N≥2, new1(X,Y,N).`

The body of this candidate definition is obtained by (i) projecting the constraint in clause 11 with respect to the variable `N` and the primed variables `X1` and `Y1`, and (ii) renaming the primed variables to unprimed variables. Since in *Defs* there is *an ancestor definition*, namely the root clause 10, with the predicate `new1` in the body, we apply the *widening operator*, introduced in [62], to clause 10 and clause 13, and we get the definition:

14. `new4(X,Y,N) :- X≥1, Y≥1, N≥1, new1(X,Y,N).`

(Recall that the widening operation of two clauses $c1$ and $c2$, after replacing every equality `A=B` by the equivalent conjunction `A≥B, A≤B`, returns the atomic constraints of clause $c1$ which are implied by the constraint of clause $c2$.)

At this point, we do *not* introduce clause 14 (as we would do if we perform a usual generalization using widening alone, as indicated in [58, 62]), but we apply our *constrained generalization*, which imposes the addition of some extra constraints to the body of clause 14, as we now explain.

With each predicate `new`k we associate a set of constraints, called the *regions for* `new`k, which are all the *atomic constraints* on the unprimed variables (that is, the variables in the heads of the clauses) occurring in any one of the clauses for `new`k in the CLP program `V` consisting of clauses 1–8. Now, let `newq(...) :- d, new`k be the candidate definition (clause 13, in our case). Then, we add to the body of the generalized definition obtained by widening, say `newp(...) :- c, new`k, (clause 14, in our case), all *negated regions for* `new`k which are implied by `d`.

In our example, the regions for `new1` are: `X<N`, `X≥1`, `X≥N`, `X≤0`, `X<Y` (see clauses 2, 3, 4 and 8) and the negated regions are, respectively: `X≥N`, `X<1`, `X<N`, `X>0`, `X≥Y`. The negated regions implied by the constraint `X=2,Y=2,N≥2`, occurring in the body of the candidate clause 13, are: `X>0` and `X≥Y`.

Thus, instead of clause 14, we introduce the following clause 15 (we wrote neither `X>0` nor `X≥1` because those constraints are implied by `X≥Y,Y≥1`):

```
15. new4(X,Y,N) :- X≥Y, Y≥1, N≥1, new1(X,Y,N).
```

and we say that clause 15 has been obtained by constrained generalization from clause 13. Clause 15 is placed in *Defs* as a child of clause 10, as clause 11 has been derived by unfolding clause 10.
Now, it remains to fold clause 12 and in order to do so, we consider the following candidate definition:

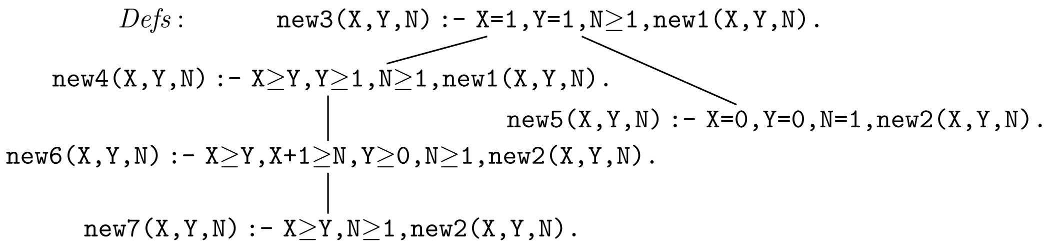
```
16. new5(X,Y,N) :- X=0, Y=0, N=1, new2(X,Y,N).
```

Clause 16 is placed in *Defs* as a child of clause 10, as clause 12 has been derived by unfolding clause 10. We do not make any generalization of this clause, because no definition with `new2` in its body occurs as an ancestor of clause 16 in *Defs*.

Now, we consider the last two definition clauses we have introduced, that is, clauses 15 and 16. First, we deal with clause 15. Starting from that clause, we perform a sequence of unfolding-definition steps similar to the sequence we have described above. During this sequence of steps, we introduce two predicates, `new6` and `new7` (see the definition clauses for those predicates in Figure 5), for performing the required folding steps.
Then, we deal with clause 16. Again, starting from that clause we perform a sequence of unfolding-definition steps. By unfolding clause 16 w.r.t. `new2` we get an empty set of clauses for `new5`. Then, we also delete clause 12, which should be folded with definition 16, because there are no clauses for `new5`.

Eventually, we get the program `S` made out of the following folded clauses:

```
17. incorrect:- X=1, Y=1, N≥1, new3(X,Y,N).
18. new3(X,Y,N) :- X=1, Y=1, N≥2, X1=2, Y1=2, new4(X1,Y1,N).
19. new4(X,Y,N) :- X≥Y, X<N, Y>0, X1=X+1, Y1=Y+1, new4(X1,Y1,N).
20. new4(X,Y,N) :- X≥Y, X≥N, Y>0, N>0, X1=X-1, Y1=Y-1, new6(X1,Y1,N).
21. new6(X,Y,N) :- X>0, X≥Y, X≥N-1, Y≥0, N>0,X1=X-1,Y1=Y-1,new7(X1,Y1,N).
```



```
Defs:        new3(X,Y,N) :- X=1,Y=1,N≥1,new1(X,Y,N).
new4(X,Y,N) :- X≥Y,Y≥1,N≥1,new1(X,Y,N).
                          new5(X,Y,N) :- X=0,Y=0,N=1,new2(X,Y,N).
new6(X,Y,N) :- X≥Y,X+1≥N,Y≥0,N≥1,new2(X,Y,N).
new7(X,Y,N) :- X≥Y,N≥1,new2(X,Y,N).
```

Figure 5: The definition tree *Defs*.

22. `new7(X,Y,N) :- X>0, X≤Y, N>0, X1=X-1, Y1=Y-1, new7(X1,Y1,N).`

This concludes the Verification Conditions Transformation step.

Now, we can perform the Verification Conditions Analysis step of our method. This phase terminates immediately because in `S` there are no constrained facts (that is, clauses whose bodies consist of constraints only) and $M(\mathtt{S})$ is the empty set. Thus, `incorrect` $\notin M(\mathtt{S})$ and we conclude that the imperative program `Prog` is correct with respect to φ_{init} and φ_{error}.

One can verify that if we were to do the generalization of Step 3 using the widening technique alone (without the constrained generalization), we could not derive a program that allows us to prove correctness, because during Step 4 the execution of the *BottomUp* procedure does not terminate. □

4.2 The Specialization Strategy

In this section we present the specialization strategy $\mathit{Specialize}_{prop}$ shown in Figure 6. Initially, $\mathit{Specialize}_{prop}$ considers the clauses of the form:

$$\mathtt{incorrect{:-}\,c_1(X)\,,A_1(X)}, \ldots, \mathtt{incorrect{:-}\,c_{\mathit{j}}(X)\,,A_{\mathit{j}}(X)} \qquad (\dagger)$$

where, for $1 \leq i \leq j$, $\mathtt{c}_i$ is either an atom or a constraint and $\mathtt{A}_i$ is a atom.

The Unfolding phase consists in unfolding a clause C with respect to the leftmost atom in its body. It makes use of the *Unf* function which takes as input a clause D and an atom `A`, and returns as output a set *SpC* of satisfiable clauses

Input: Program P (either `V` or `R`).
Output: Program `S` such that `incorrect` $\in M(P)$ iff `incorrect` $\in M($`S`$)$.

```
INITIALIZATION:
S := $\emptyset$;
$\mathit{InCls} := \{$ incorrect :- c$_1$(X), A$_1$(X), ..., incorrect :- c$_j$(X), A$_j$(X) $\}$;
$\mathit{Defs} := \emptyset$;
  ($\alpha$) while in $\mathit{InCls}$ there is a clause $C$ that is not a constrained fact do
    UNFOLDING:
      $\mathit{SpC} := \mathit{Unf}(C, \mathtt{A})$, where A is the leftmost atom in the body of $C$;
    CLAUSE REMOVAL:
      ($\alpha$1) while in $\mathit{SpC}$ there are two distinct clauses $E_1$ and $E_2$ such that
                  $E_1$ subsumes $E_2$ do
            $\mathit{SpC} := \mathit{SpC} - \{E_2\}$;
          end-while;
    DEFINITION INTRODUCTION:
      ($\alpha$2) while in $\mathit{SpC}$ there is a clause $E$ that is not a constrained fact
                  and cannot be folded using a definition in $\mathit{Defs}$ do
            $G := \mathit{Gen}(E, \mathit{Defs})$;
            $\mathit{Defs} := \mathit{Defs} \cup \{\mathit{child}(G, C)\}$;
            $\mathit{InCls} := \mathit{InCls} \cup \{G\}$;
          end-while;
    $\mathit{InCls} := \mathit{InCls} - \{C\}$;
    S := S $\cup$ $\mathit{SpC}$;
  end-while;
FOLDING:
  ($\beta$) while in S there is a clause $E$ that can be folded
              by a clause $D$ in $\mathit{Defs}$ do
          S := (S $- \{E\}) \cup \{F\}$, where $F$ is derived by folding $E$ using $D$;
        end-while;
Remove from S all clauses for predicates on which incorrect does not depend.
```

Figure 6: The *Specialize*$_{prop}$ Procedure.

derived from D by a single application of the unfolding rule (see Definition 1), which consists in: (i) replacing an atom $\mathtt{A}$ occurring in the body of a clause by the bodies of the clauses in P whose head is unifiable with $\mathtt{A}$, and (ii) applying the unifying substitution.

At the end of the *Unf* procedure, CLAUSE REMOVAL removes subsumed clauses.

The specialization strategy proceeds to the DEFINITION INTRODUCTION phase and terminates when no new definitions are needed for performing the subsequent FOLDING phase. Unfortunately, an uncontrolled application of the DEFINITION INTRODUCTION procedure may lead to the introduction of infinitely many new definitions, thereby causing the nontermination of the specialization procedure. In order to deal with this potential nontermination issue we introduce a subsidiary procedure, called *Gen*, which introduces new definitions and is parametric with respect to generalization operators.

In the following section we will define suitable generalization operators which guarantee the introduction of finitely many new definitions.

4.2.1 Generalization Operators

In this section we define some generalization operators which are used to ensure the termination of the specialization strategy and, as mentioned in the introduction, we also introduce *constrained* generalization operators that generalize the constraints occurring in a candidate definition and, by adding suitable extra constraints, have the objective of preventing that the set of clauses generated by unfolding the generalized definition is larger than the set of clauses generated by unfolding the candidate definition. In this sense we say the objective of constrained generalization is to preserve the branching behaviour of the candidate definitions.

More generalization operators used for the specialization of logic programs and also constraint logic programs can be found in [58, 62, 102, 101, 118].

We will consider linear constraints $\mathcal{C}$ over the set $\mathcal{R}$ of the real numbers. The set $\mathcal{C}$ is the minimal set of constraints which: (i) includes all atomic constraints of the form either $\mathtt{p}_1 \leq \mathtt{p}_2$ or $\mathtt{p}_1 < \mathtt{p}_2$, where $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$ and integer coefficients, and (ii) is closed under conjunction (which we denote by ',' and also by '$\wedge$'). An equation $\mathtt{p}_1 = \mathtt{p}_2$ stands for $\mathtt{p}_1 \leq \mathtt{p}_2 \wedge \mathtt{p}_2 \leq \mathtt{p}_1$. The projection of a constraint $\mathtt{c}$ onto a tuple $\mathtt{X}$ of variables, denoted $\mathit{project}(\mathtt{c}, \mathtt{X})$, is a constraint such that $\mathcal{R} \models \forall \mathtt{X}\ (\mathit{project}(\mathtt{c}, \mathtt{X}) \leftrightarrow \exists \mathtt{Y} \mathtt{c})$, where Y is the tuple of variables occurring in $\mathtt{c}$ and not in $\mathtt{X}$.

In order to introduce the notion of a generalization operator we need the following definition [50].

Definition 7 (Well-Quasi Ordering $\precsim$). A *well-quasi ordering* (or *wqo*, for short) on a set S is a reflexive, transitive relation $\precsim$ on S such that, for every infinite sequence $e_0 e_1 \ldots$ of elements of S, there exist i and j such that $i<j$ and $e_i \precsim e_j$. Given e_1 and e_2 in S, we write $e_1 \approx e_2$ if $e_1 \precsim e_2$ and $e_2 \precsim e_1$. A wqo $\precsim$ is *thin* iff for all $e \in S$, the set $\{e' \in S \mid e \approx e'\}$ is finite.

The use of a thin wqo guarantees that during the *Specialize*$_{prop}$ procedure each definition can be generalized a finite number of times only, and thus the termination of the procedure is guaranteed.

The thin wqo *Maxcoeff*, denoted by $\precsim_M$, compares the maximum absolute values of the coefficients occurring in polynomials. It is defined as follows. For any atomic constraint $\mathtt{a}$ of the form $\mathtt{p} < 0$ or $\mathtt{p} \leq 0$, where $\mathtt{p}$ is $q_0 + q_1\mathtt{X}_1 + \ldots + q_k\mathtt{X}_k$, we define *maxcoeff*($\mathtt{a}$) to be $\max\{|q_0|, |q_1|, \ldots, |q_k|\}$. Given two atomic constraints $\mathtt{a}_1$ of the form $\mathtt{p}_1 < 0$ and $\mathtt{a}_2$ of the form $\mathtt{p}_2 < 0$, we have that $\mathtt{a}_1 \precsim_M \mathtt{a}_2$ iff *maxcoeff*($\mathtt{a}_1$) $\leq$ *maxcoeff*($\mathtt{a}_2$).

Similarly, if we are given the atomic constraints $\mathtt{a}_1$ of the form $\mathtt{p}_1 \leq 0$ and $\mathtt{a}_2$ of the form $\mathtt{p}_2 \leq 0$. Given two constraints $\mathtt{c}_1 \equiv \mathtt{a}_1, \ldots, \mathtt{a}_m$, and $\mathtt{c}_2 \equiv \mathtt{b}_1, \ldots, \mathtt{b}_n$, we have that $\mathtt{c}_1 \precsim_M \mathtt{c}_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $\mathtt{a}_i \precsim_M \mathtt{b}_j$. For example, we have that:
(i) $(1-2\mathtt{X}_1 < 0) \;\precsim_M\; (3+\mathtt{X}_1 < 0)$,
(ii) $(2-2\mathtt{X}_1+\mathtt{X}_2 < 0) \;\precsim_M\; (1+3\mathtt{X}_1 < 0)$, and
(iii) $(1+3\mathtt{X}_1 < 0) \;\not\precsim_M\; (2-2\mathtt{X}_1+\mathtt{X}_2 < 0)$.

Definition 8 (Generalization Operator $\ominus$). Let $\precsim$ be a thin wqo on the set $\mathcal{C}$ of constraints. A function $\ominus$ from $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$ is a *generalization operator* with respect to $\precsim$ if, for all constraints $\mathtt{c}$ and $\mathtt{d}$, we have: (i) $\mathtt{d} \sqsubseteq \mathtt{c} \ominus \mathtt{d}$, and (ii) $\mathtt{c} \ominus \mathtt{d} \precsim \mathtt{c}$.

A trivial generalization operator is defined as $\mathtt{c} \ominus \mathtt{d} = \mathtt{true}$, for all constraints $\mathtt{c}$ and $\mathtt{d}$ (without loss of generality we assume that $\mathtt{true} \precsim \mathtt{c}$ for every constraint $\mathtt{c}$).

Definition 8 generalizes several operators proposed in the literature, such as the widening operator [30] and the *most specific generalization* operator [102, 132].

Other generalization operators defined in terms of relations and operators on constraints such as *widening* and *convex-hull* which have been proposed for the static analysis of programs [30, 34] and also applied to the specialization of constraint logic programs (see, for instance, [62, 118]). These generalization operators have been extensively studied in the above cited papers.

Here we define some generalization operators which have been used in the experiments we have performed (see also [62]).

• (W) Given any two constraints $\mathtt{c} \equiv \mathtt{a}_1, \ldots, \mathtt{a}_m$, and $\mathtt{d}$, the operator *Widen*, denoted $\ominus_W$, returns the constraint $\mathtt{a}_{i1}, \ldots, \mathtt{a}_{ir}$, such that $\{\mathtt{a}_{i1}, \ldots, \mathtt{a}_{ir}\} = \{\mathtt{a}_h \mid 1 \leq h \leq m \text{ and } \mathtt{d} \sqsubseteq \mathtt{a}_h\}$. Thus, *Widen* returns all atomic constraints of $\mathtt{c}$ that are entailed by $\mathtt{d}$ (see [30] for a similar widening operator used in static program analysis). The operator $\ominus_W$ is a generalization operator w.r.t. the thin wqo $\precsim_M$.

• (WM) Given any two constraints $\mathtt{c} \equiv \mathtt{a}_1, \ldots, \mathtt{a}_m$, and $\mathtt{d} \equiv \mathtt{b}_1, \ldots, \mathtt{b}_n$, the operator *WidenMax*, denoted $\ominus_{WM}$, returns the conjunction $\mathtt{a}_{i1}, \ldots, \mathtt{a}_{ir}, \mathtt{b}_{j1}, \ldots, \mathtt{b}_{js}$, where: (i) $\{\mathtt{a}_{i1}, \ldots, \mathtt{a}_{ir}\} = \{\mathtt{a}_h \mid 1 \leq h \leq m \text{ and } \mathtt{d} \sqsubseteq \mathtt{a}_h\}$, and (ii) $\{\mathtt{b}_{j1}, \ldots, \mathtt{b}_{js}\} = \{\mathtt{b}_k \mid 1 \leq k \leq n \text{ and } \mathtt{b}_k \precsim_M \mathtt{c}\}$.

The operator *WidenMax* is a generalization operator w.r.t. the thin wqo $\precsim_M$. It is similar to *Widen* but, together with the atomic constraints of $\mathtt{c}$ that are entailed by $\mathtt{d}$, it returns also the conjunction of a subset of the atomic constraints of $\mathtt{d}$.

Next we define a generalization operator by using the *convex hull* operator, which is often used in static program analysis [34].

• (CH) The *convex hull* of two constraints $\mathtt{c}$ and $\mathtt{d}$ in $\mathcal{C}$, denoted by $ch(\mathtt{c}, \mathtt{d})$, is the least (w.r.t. the $\sqsubseteq$ ordering) constraint $\mathtt{h}$ in $\mathcal{C}$ such that $\mathtt{c} \sqsubseteq \mathtt{h}$ and $\mathtt{d} \sqsubseteq \mathtt{h}$. (Note that $ch(\mathtt{c}, \mathtt{d})$ is unique up to equivalence of constraints.)

• ($CHWM$) Given any two constraints $\mathtt{c}$ and $\mathtt{d}$, we define the operator *CHWidenMax*, denoted $\ominus_{CHWM}$, as follows: $\mathtt{c} \ominus_{CHWM} \mathtt{d} = \mathtt{c} \ominus_{WM} ch(\mathtt{c}, \mathtt{d})$. The operator $\ominus_{CHWM}$ is a generalization operator w.r.t. the thin wqo $\precsim_M$.

CHWidenMax returns the conjunction of a subset of the atomic constraints of $\mathtt{c}$ and a subset of the atomic constraints of $ch(\mathtt{c}, \mathtt{d})$.

Constrained Generalization

Now we describe a method for deriving, from any given generalization operator $\ominus$, a new version of that operator, denoted $\ominus_{cns}$, which adds some extra constraints and still is a generalization operator. The operator $\ominus_{cns}$ is called the *constrained generalization operator derived from* $\ominus$.

In order to specify the constrained generalization operator we need the following notions.

Let P be the input program of the *Specialize*$_{prop}$ procedure. For any constraint $\mathtt{d}$ and atom $\mathtt{A}$, we define the *unfeasible clauses* for the pair $(\mathtt{d}, \mathtt{A})$, denoted $\mathit{UnfCl}(\mathtt{d},\mathtt{A})$, to be the set $\{(\mathtt{H}_1 \,\mathtt{:\text{-}}\, \mathtt{c}_1 \,\mathtt{,} \mathtt{G}_1), \ldots, (\mathtt{H}_m \,\mathtt{:\text{-}}\, \mathtt{c}_m \,\mathtt{,} \mathtt{G}_m)\}$, of (renamed apart) clauses of P such that, for $i = 1, \ldots, m$, $\mathtt{A}$ and $\mathtt{H}_i$ are unifiable via the most general unifier ϑ_i and $(\mathtt{d} \wedge \mathtt{c}_i)\,\vartheta_i$ is unsatisfiable.

The *head constraint* of a clause of the form `H :- c, A` is the constraint $\mathit{project}(\mathtt{c}, \mathtt{X})$, where X is the tuple of variables occurring in $\mathtt{H}$. For any atomic constraint $\mathtt{a}$, $\mathit{neg}(\mathtt{a})$ denotes the negation of $\mathtt{a}$ defined as follows: $\mathit{neg}(\mathtt{p} < 0)$ is $-\mathtt{p} \leq 0$ and $\mathit{neg}(\mathtt{p} \leq 0)$ is $-\mathtt{p} < 0$. Given a set C of clauses, we define the set of the *negated regions* of C, denoted $\mathit{NegReg}(C)$, as follows:

$$\mathit{NegReg}(C) = \{\mathit{neg}(\mathtt{a}) \mid \mathtt{a} \text{ is an atomic constraint of a head constraint of a clause in } C\}.$$

For any constraint $\mathtt{d}$ and atom $\mathtt{A}$, we define the following constraint:

$$\mathit{cns}(\mathtt{d},\mathtt{A}) = \bigwedge\{\, \mathtt{r} \mid \mathtt{r} \in \mathit{NegReg}(\,\mathit{UnfCl}(\mathtt{d},\mathtt{A})) \;\wedge\; \mathtt{d} \sqsubseteq \mathtt{r}\}.$$

We have that $\mathtt{d} \sqsubseteq \mathit{cns}(\mathtt{d},\mathtt{A})$. Now, let $\ominus$ be a generalization operator with respect to the thin wqo $\precsim$. We define the constrained generalization operator derived from $\ominus$, as follows:

$$\ominus_{\mathit{cns}}(\mathtt{c},\mathtt{d},\mathtt{A}) = (\mathtt{c} \ominus \mathtt{d}) \wedge \mathit{cns}(\mathtt{d},\mathtt{A}).$$

Now we show that $\ominus_{\mathit{cns}}$ is indeed a generalization operator w.r.t. the thin wqo $\precsim_B$ we now define. Given a finite set B of (non necessarily atomic) constraints, a constraint $\mathtt{c}_1 \wedge \ldots \wedge \mathtt{c}_n$, where $\mathtt{c}_1, \ldots, \mathtt{c}_n$ are atomic, and a constraint $\mathtt{d}$, we define the binary relation $\precsim_B$ on constraints as follows: $\mathtt{c}_1 \wedge \ldots \wedge \mathtt{c}_n \precsim_B \mathtt{d}$ iff *either* (i) $(\mathtt{c}_1 \wedge \ldots \wedge \mathtt{c}_n) \precsim \mathtt{d}$, *or* (ii) there exists $i \in \{1, \ldots, n\}$ such that $\mathtt{c}_i \in B$ and $(\mathtt{c}_1 \wedge \ldots \wedge \mathtt{c}_{i-1} \wedge \mathtt{c}_{i+1} \wedge \ldots \wedge \mathtt{c}_n) \precsim_B \mathtt{d}$. It can be shown that $\precsim_B$ is a thin wqo.

We observe that, for all constraints $\mathtt{c}$, $\mathtt{d}$, and all atoms $\mathtt{A}$: (i) since $\mathtt{d} \sqsubseteq \mathtt{c} \ominus \mathtt{d}$ and $\mathtt{d} \sqsubseteq \mathit{cns}(\mathtt{d},\mathtt{A})$, then also $\mathtt{d} \sqsubseteq \ominus_{\mathit{cns}}(\mathtt{c},\mathtt{d},\mathtt{A})$, and (ii) by definition of $\precsim_B$, for all constraints e, if $\mathtt{c} \ominus \mathtt{d} \precsim \mathtt{e}$, then $\ominus_{\mathit{cns}}(\mathtt{c},\mathtt{d},\mathtt{A}) \precsim_B \mathtt{e}$, where $B = \mathit{NegReg}(P)$.

Thus, we have the following result.

Proposition 1. For any program $P \cup \{\gamma_0\}$ given as input to the $\mathit{Specialize}_{\mathit{vcg}}$ procedure, for any atom $\mathtt{A}$, the operator $\ominus_{\mathit{cns}}(\_,\_,\mathtt{A})$ is a generalization operator with respect to the thin well-quasi ordering $\precsim_B$, where $B = \mathit{NegReg}(P)$.

4.2.2 Generalization Strategy

The $\mathit{Specialize}_{\mathit{prop}}$ procedure introduces new *definitions* by using the subsidiary *Gen* strategy which, given a clause E and a set *Defs* of definitions, yields a new definition clause G.

Definition 9 (Generalization Strategy). Let E be a clause of the form `H(X) :- e(X,X1), Q(X1)`, where `X` and `X1` are tuples of variables, `e(X,X1)` is a constraint, and `Q(X1)` is an atom. Let *Defs* be a set of definitions. Then, $\mathit{Gen}(E, \mathit{Defs})$ is a clause G: `newp(X) :- g(X), Q(X)`, such that: (i) `newp` is a new predicate symbol, and (ii) `e(X,X1)` $\sqsubseteq$ `g(X1)`.

For any infinite sequence $E_1, E_2, \ldots$ of clauses, let $G_1, G_2, \ldots$ be a sequence of clauses constructed as follows: (1) $G_1 = \mathit{Gen}(E_1, \emptyset)$, and (2) for every $i > 0$, $G_{i+1} = \mathit{Gen}(E_{i+1}, \{G_1, \ldots, G_i\})$. We assume that the sequence $G_1, G_2, \ldots$ *stabilizes*, that is, there exists an index k such that, for every $i > k$, G_i is equal, modulo the head predicate name, to a clause in $\{G_1, \ldots, G_k\}$.

In order to control the application of the generalization operators, we follow an approach which is similar to one considered in the context of partial deduction [101, 109]. In particular, the *Gen* strategy makes use of a set *Defs* of definitions arranged as a *forest* of *trees* whose roots are among the clauses (†) considered at the beginning of the specialization process. Each new definition G introduced during specialization determines a new node $\mathit{child}(G, C)$ of a tree which is placed as a child of definition C if G is introduced to fold a clause derived by unfolding C.

Figure 7 shows the generalization strategy used in $\mathit{Specialize}_{prop}$. In Section 4.3 we will see the specific (constrained) generalization operators ($\ominus_{cns}$) $\ominus$ which will be used in the experimental evaluation.

Input: A clause E and a set *Defs* of definitions structured as a tree.
Output: A new definition G.

```
Let $E$ be a clause of the form q :- d, A(X) and $d_X$ = $\mathit{project}$(d, X).
  $\mathit{if}$ there exists a clause $D$ in $\mathit{Defs}$ such that:
 (i)  $D$ is of the form p(Y) :- c, A(Y), and
 (ii) $D$ is the most recent ancestor of $E$ in $\mathit{Defs}$
      such that A(Y) is a variant of A(X)
  $\mathit{then}$ $G :=$ newp(X) :- g, A where g is either c $\ominus$ d or $\ominus_{cns}$(c, $d_X$, A)
  $\mathit{else}$ $G :=$ newp(X) :- $d_X$, A
```

Figure 7: The generalization strategy *Gen*(*E*,*Defs*) of $\mathit{Specialize}_{prop}$

4.2.3 Termination and Soundness of the Specialization Strategy

The following theorem establishes the termination and soundness of the *Specialize$_{prop}$* strategy.

Theorem 4 (Termination and Correctness of the *Specialize$_{prop}$* strategy). (i) The *Specialize$_{prop}$* procedure always terminates. (ii) Let program `S` be the output of the *Specialize$_{prop}$* procedure. Then `incorrect` $\in M($`V`$)$ iff `incorrect`$\in M($`S`$)$.

Proof. (i) Since the UNFOLDING subsidiary procedure, the CLAUSE REMOVAL while-loop ($\alpha 1$), the DEFINITION INTRODUCTION while-loop ($\alpha 2$), and the FOLDING while-loop (β) clearly terminate, we are left with showing that the first, outermost while-loop (α) terminates, that is, a finite number of new predicate definitions is added to *InCls* by DEFINITION INTRODUCTION. This finiteness is guaranteed by the following facts:
(1) all new predicate definitions are introduced by the *Gen* strategy,
(2) by Definition 9 the set of all new predicate definitions generated by a sequence of applications of a generalization operator is finite, modulo the head predicate names, and
(3) no two new predicate definitions that are equal modulo the head predicate name are introduced by DEFINITION INTRODUCTION (indeed, DEFINITION INTRODUCTION introduces a new predicate definition only if the definitions already present in *Defs* cannot be used to fold clause E).

(ii) see the proof of point (ii) of Theorem 3.

□

4.3 Experimental Evaluation

In this section we present some preliminary results obtained by applying our Software Model Checking method to some benchmark programs taken from the literature. The results show that our approach is viable and competitive with the state-of-the-art software model checkers.

Programs `ex1`, `f1a`, `f2`, and `interp` have been taken from the benchmark set of DAGGER [75]. Programs `substring` and `tracerP` are taken from [91] and [86], respectively. Programs `doubleLoop` and `singleLoop` have been introduced to illustrate the constrained generalization strategy. Finally, `selectSort` is an encoding of the Selection sort algorithm where references to arrays have been abstracted away to perform array bounds checking. The source code of all the above programs is available at `http://map.uniroma2.it/smc/`.

The experiments have been performed by using the VeriMAP software model checker (see Chapter 8) that implements our verification method. We have also run three state-of-the-art CLP-based software model checkers on the same set of programs, and we have compared their performance with that of our model checker. In particular, we have used: (i) ARMC [123], (ii) HSF(C) [73], and (iii) TRACER [85]. ARMC and HSF(C) are CLP-based software model checkers which implement the CEGAR technique. TRACER is a CLP-based model checker which uses Symbolic Execution (SE) for the verification of partial correctness properties of sequential C programs using approximated preconditions or approximated postconditions.

Table 4.1 reports the results of our experimental evaluation which has been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

In Columns W and $CHWM$ we report the results obtained by the MAP system when using the procedure presented in Section 4.2.1 and the generalization operators *Widen* and *CHWidenMax* [62], respectively. In Columns W_{cns} and $CHWM_{cns}$ we report the results for the constrained versions of those generalization operators, called $Widen_{cns}$ and $CHWidenMax_{cns}$, respectively. In the remaining columns we report the results obtained by ARMC, HSF(C), and TRACER using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options, respectively.

On the selected set of examples, we have that the MAP system with the $CHWidenMax_{cns}$ is able to verify 9 properties out of 9, while the other tools do not exceed 7 properties. Also the verification time is generally comparable to that of the other tools, and it is not much greater than that of the fastest tools. Note that there are two examples (`doubleLoop` and `singleLoop`) where constrained generalization operators based on widening and convex-hull are strictly more powerful than the corresponding operators which are not constrained.

We also observe that the use of a constrained generalization operator usually causes a very small increase of the verification time with respect to the non-constrained counterparts, thus making constrained generalization a promising technique that can be used in practice for software verification.

In Table 4.2 we present in some more detail the time taken for proving the properties of interest by using our method for software model checking with the generalization operators *Widen* (Column W) and *CHWidenMax* (Column $CHWM$), and the constrained generalization operators derived from them

Program	MAP				ARMC	HSF(C)	TRACER	
	W	W_{cns}	$CHWM$	$CHWM_{cns}$			$SPost$	$WPre$
`ex1`	1.08	1.09	1.14	1.25	0.18	0.21	∞	1.29
`f1a`	∞	∞	0.35	0.36	∞	0.20	$\perp$	1.30
`f2`	∞	∞	0.75	0.88	∞	0.19	∞	1.32
`interp`	0.29	0.29	0.32	0.44	0.13	0.18	∞	1.22
`doubleLoop`	∞	0.33	0.33	0.33	∞	0.19	∞	∞
`selectSort`	4.34	4.70	4.59	5.57	0.48	0.25	∞	∞
`singleLoop`	∞	∞	∞	0.26	∞	∞	$\perp$	1.28
`substring`	88.20	171.20	5.21	5.92	931.02	1.08	187.91	184.09
`tracerP`	0.11	0.12	0.11	0.12	∞	∞	1.15	1.28

Table 4.1: Time (in seconds) taken for performing model checking. '∞' means 'no answer within 20 minutes', and '$\perp$' means 'termination with error'.

$Widen_{cns}$ (Column W_{cns}) and $CHWidenMax_{cns}$ (Column $CHWM_{cns}$), respectively.

Columns Step 1-2, Step 3, and Step 4 show the time required for performing the corresponding Step, respectively, of our Software Model Checking method presented in Section 4.1. The sum of these three times for each phase is reported in Column *Tot*.

4.4 Related Work

The specialization of logic programs and constraint logic programs has also been used in techniques for the verification of infinite state reactive systems [59, 62, 61, 103]. By using these techniques one may verify properties of Kripke structures, instead of properties of imperative programs as we did here. In [103] the authors do not make use of constraints, which are a key ingredient of the technique we present here. In [61, 62, 103] program specialization is combined with the computation of the least model of the specialized program, or the computation of an overapproximation of the least model via abstract interpretation.

The use of program specialization for the verification of properties of imperative programs is not novel. It has been investigated, for instance, in [119]. In that paper a CLP interpreter for the operational semantics of a simple imperative language is specialized with respect to the input program to be verified. Then, a static analyzer for CLP programs is applied to the residual program for computing invariants (that is, overapproximations of the behavior) of the input imperative program. These invariants are used in the proof of the properties of interest. Unlike [119], our verification method does not perform any static analysis phase separated from the specialization phase and, instead, we discover program invariants during the specialization process by applying suitable generalization operators. These operators are defined in terms of operators and relations on constraints such as widening and convex-hull [30, 34, 62]. As in [119], we also use program specialization to perform the so-called removal of the interpreter, but in addition, in this paper we use specialization for propagating the information about the constraints in the initial configurations. In particular, the CLP program is specialized with respect to the property to be verified, by using constrained generalization operators which have the objective of preserving, if possible, the branching behaviour of the definitions to be generalized. In this way we may avoid loss of precision, and at the same time, we enforce the termination of the specialization process Step 3.

The idea of constrained generalization which has the objective of preserving

Program	Step 1-2	W			W_{cns}			$CHWM$			$CHWM_{cns}$		
		Step 3	Step 4	Tot	Step 3	Step 4	Tot	Step 3	Step 4	Tot	Step 3	Step 4	Tot
`ex1`	1.02	0.05	0.01	1.08	0.07 $\triangleleft$	0	1.09	0.11	0.01	1.14	0.23 $\triangleleft$	0	1.25
`f1a`	0.35	0.01	∞	∞	0.01	∞	∞	0 $\triangleleft$	0	0.35	0.01 $\triangleleft$	0	0.36
`f2`	0.71	0.03	∞	∞	0.13	∞	∞	0.03 $\triangleleft$	0.01	0.75	0.17 $\triangleleft$	0	0.88
`interp`	0.27	0.01	0.01	0.29	0.02 $\triangleleft$	0	0.29	0.04	0.01	0.32	0.17 $\triangleleft$	0	0.44
`doubleLoop`	0.31	0.01	∞	∞	0.02 $\triangleleft$	0	0.33	0.02 $\triangleleft$	0	0.33	0.02 $\triangleleft$	0	0.33
`selectSort`	4.27	0.06	0.01	4.34	0.43 $\triangleleft$	0	4.70	0.3	0.02	4.59	1.3 $\triangleleft$	0	5.57
`singleLoop`	0.22	0.02	∞	∞	0.02	∞	∞	0.03	∞	∞	0.04 $\triangleleft$	0	0.26
`substring`	0.24	0.01	87.95	88.20	0.02	170.94	171.2	4.96 $\triangleleft$	0.01	5.21	5.67 $\triangleleft$	0.01	5.92
`tracerP`	0.11	0 $\triangleleft$	0	0.11	0.01 $\triangleleft$	0	0.12	0 $\triangleleft$	0	0.11	0.01 $\triangleleft$	0	0.12

Table 4.2: Time (in seconds) taken for performing software model checking with the MAP system. '∞' means 'no answer within 20 minutes'. Times marked by '$\triangleleft$' are relative to the programs obtained after Step 3 and have no constrained facts (thus, for those programs the times of Step 4 are very small ($\leq 0.01\,s$)).

the branching behaviour of a clause, is related to the technique for preserving *characteristic trees* while applying abstraction during partial deduction [104]. Indeed, a characteristic tree provides an abstract description of the tree generated by unfolding a given goal, and abstraction corresponds to generalization. However, the partial deduction technique considered in [104] is applied to ordinary logic programs (not CLP programs) and constraints such as equations and inequations on finite terms, are only used in an intermediate phase.

In order to get a conservative model of a program, different generalization operators have been introduced in the literature. In particular, in [15] the authors introduce the *bounded widen* operator $\mathtt{c} \nabla_B \mathtt{d}$, defined for any given constraint $\mathtt{c}$ and $\mathtt{d}$ and any set B of constraints. This operator, which improves the precision of the *widen* operator introduced in [30], has been applied in the verification of synchronous programs and linear hybrid systems. A similar operator $\mathtt{c} \nabla_B \mathtt{d}$, called *widening up to* B, has been introduced in [78]. In this operator the set B of constraints is statically computed once the system to be verified is given. There is also a version of that operator, called *interpolated widen*, in which the set B is dynamically computed [75] by using the interpolants which are derived during the counterexample analysis.

Similarly to [15, 34, 75, 78], the main objective of the constrained generalization operators introduced in this paper is the improvement of precision during program specialization. In particular, this generalization operator, similar to the bounded widen operator, limits the possible generalizations on the basis of a set of constraints defined by the CLP program obtained as output of Step 3. Since this set of constraints which limits the generalization depends on the output of Step 3, our generalization is more flexible than the one presented in [15]. Moreover, our generalization operator is more general than the classical widening operator introduced in [30]. Indeed, we only require that the set of constraints which have a non-empty intersection with the generalized constraint $\mathtt{c} \ominus \mathtt{d}$, are entailed by $\mathtt{d}$.

CHAPTER 5

Iterated Program Specialization

We have shown that program specialization can be used not only as a preprocessing step to generate verification conditions, but also as a means of analysis on its own. Indeed, by specializing the CLP program `V` with respect to the constraints characterizing the input values of `Prog`, in some cases one can derive a new CLP program `S` whose least model $M(\mathtt{S})$ can be computed in finite time because `S` can be represented by a finite (possibly empty) set of constraints. Thus, in these cases it is possible to verify whether or not `Prog` is correct with respect to φ_{init} and φ_{error} by simply inspecting that model.

However, due to the undecidability of partial correctness, it is impossible to devise a specialization technique that always terminates and produces a specialized program whose least model can be finitely computed. Thus, the best one can do is to propose a verification technique based on some heuristics and show that it works well in practice.

In this chapter we present a method, called *iterated specialization*, which extends the method proposed in Chapter 4 and it is based on the repeated application of program specialization. By iterated specialization we produce a sequence of programs of the form $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \ldots$ Each program specialization step terminates and has the effect of modifying the structure of the CLP program (and consequently the structure of the corresponding set of verification conditions) by explicitly adding new constraints that denote invariants of the computation. The effect of the iterated specialization is the propagation of these constraints from one program version to the next, and since each new iteration starts from the output of the previous one, we can refine program analysis and possibly increase the level of precision.

The use of iterated specialization avoids the least model construction performed by the technique presented in the previous Chapter 4 after program specialization.

Iterated specialization terminates at step k, if a *lightweight analysis* based on a simple inspection of program $\mathtt{S}_k$ is able to decide whether or not the given incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ `Prog` $\{\!\{\varphi_{error}\}\!\}$ holds. While each transformation step is guaranteed to terminate, due to undecidability of the partial correctness, the overall process may not terminate.

In order to validate the heuristics used in our verification method from an experimental point of view, we have used our prototype verification system called VeriMAP (see Chapter 8). We have performed verification tests on a significant set of over 200 programs taken from various publicly available benchmarks. The precision of our system, that is the ratio of the successfully verified programs over the total number of programs, is about 85 percent. We have also compared the results we have obtained using the VeriMAP system with the results we have obtained using other state-of-the-art software model checking systems, such as ARMC [123], HSF(C) [73], and TRACER [85]. These results show that our verification system has a considerably higher precision.

This chapter is organized as follows. In Section 5.1 we describe the overall strategy of iterated specialization, and in Section 5.2 also some specific strategies for performing the individual specialization steps. In Section 5.3 we report on the experiments we have performed by using our prototype implementation, and we compare our results with the results we have obtained using ARMC, HSF(C), and TRACER. Finally, in Section 5.4 we discuss the related work and, in particular, we compare our approach with other existing methods of software model checking.

5.1 The Verification Method

As an alternative to the construction of the least model and to standard query evaluation strategies, we present a software model checking method based on *iterated specialization*, which performs a *sequence* of program specializations, thereby producing a sequence $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \ldots$ of specialized CLP programs. During the various specializations we may apply different strategies for *propagating constraints* (either propagating forward from an initial configuration to an error configuration, or propagating backward from an error configuration to an initial configuration) and different operators (such as the widening and convex hull operators) for generalizing predicate definitions.

Iterated specialization has the objective of deriving a new CLP program $\mathtt{S}_{i+1}$

such that: (i) `incorrect` $\in M(\mathtt{S}_i)$ iff `incorrect` $\in M(\mathtt{S}_{i+1})$, and (ii) $\mathtt{S}_{i+1}$ either contains the fact `incorrect` or contains no clauses with head `incorrect`. In the former case the incorrectness triple $\{\!\{\varphi_{\mathit{init}}\}\!\}$ `Prog` $\{\!\{\varphi_{\mathit{error}}\}\!\}$ holds and the given imperative program `Prog` is *incorrect* with respect to φ_{init} and φ_{error}, while in the latter case the incorrectness triple does not h old and the given imperative program `Prog` is *correct* with respect to φ_{init} and φ_{error}.

The Software Model Checking Method

```
Input:  An incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ Prog $\{\!\{\varphi_{error}\}\!\}$ and
        the CLP program I defining the predicate incorrect.
Output: If Prog is correct with respect to $\varphi_{init}$ and $\varphi_{error}$ then 'correct'
        else 'incorrect'.

Step 1: T := $C2CLP$(Prog, $\varphi_{init}$, $\varphi_{error}$);   P := T ∪ I;
Step 2: V := $Specialize_{vcg}$(P);
Step 3: S := $Specialize_{prop}$(V);
Step 4: if $CorrectnessTest$ (S) = 'correct' then return 'correct';
        elseif $CorrectnessTest$ (S) = 'incorrect' then return 'incorrect';
        else { V := $Reverse$(S); goto Step 3 }
```

Figure 8: The Iterated Verification Strategy

The verification method is outlined in Figure 8. The given incorrectness triple $\{\!\{\varphi_{\mathit{init}}\}\!\}$ `Prog` $\{\!\{\varphi_{\mathit{error}}\}\!\}$ is processed by the CLP Translation and Verification Conditions Generation steps presented in Chapter 3.

Then, the Iterated Specialization strategy applies the procedure $\mathit{Specialize}_{\mathit{prop}}$, which propagates the constraints of the initial configuration. The constraints of the initial configuration can be propagated through the program `V` obtained after removing the interpreter, by specializing `V` itself with respect to φ_{init}, thereby deriving a new specialized program `S`.

Next, our Iterated Specialization strategy performs a lightweight analysis, called the *CorrectnessTest*, to check whether or not `incorrect` belongs to $M(\mathtt{S})$, that is, whether or not `Prog` is correct. In particular, *CorrectnessTest* checks whether `S` can be transformed into an equivalent program `Q` where one of the following conditions holds: either (i) the fact `incorrect` belongs to `Q`, hence there is a computation leading to an error configuration and the strategy halts reporting '*incorrect*', or (ii) `Q` has no constrained facts, hence no computation leads to an error configuration and the strategy halts reporting '*correct*', or

(iii) `Q` contains a clause of the form `incorrect :- G`, where `G` is not the empty goal (thus, neither (i) nor (ii) holds), and hence the strategy proceeds to the subsequent step.

In that subsequent step our strategy propagates the constraints of the error configuration. This is done by: (i) first applying the *Reverse* procedure, which, so to say, inverts the flow of computation by interchanging the roles of the initial configuration and the error configuration, and (ii) then specializing (using the procedure $\mathit{Specialize}_{prop}$ again) the 'reversed' program with respect to φ_{error}.

The strategy iterates the applications of the *Reverse* and the $\mathit{Specialize}_{prop}$ procedures until hopefully *CorrectnessTest* succeeds, thereby reporting either '*correct*' or '*incorrect*'. Obviously, due to the undecidability of partial correctness, the Iterated Specialization strategy may not terminate. However, we will show that each iteration terminates, and hence we can refine the analysis and possibly increase the level of precision by starting each new iteration from the CLP program obtained as output of the previous iteration.

5.2 The Iterated Specialization Strategy

In this section we describe the basic components required to realize the Iterated Specialization strategy, that is, (i) the $\mathit{Specialize}_{prop}$ procedure, (ii) the *CorrectnessTest* procedure, and (iii) the *Reverse* procedure.

5.2.1 Propagation of Constraints

The procedure $\mathit{Specialize}_{prop}$ specializes the CLP program `V`, obtained after the application of $\mathit{Specialize}_{vcg}$, by propagating the constraints that characterize the initial or the error configuration. (The fact that they characterize either an initial or an error configuration depends on the number of applications of the *Reverse* procedure.)

Now we describe how the unfolding and generalization steps are performed during $\mathit{Specialize}_{prop}$.

In order to guide the application of the unfolding rule during the application of $\mathit{Specialize}_{prop}$, we stipulate that every atom with a new predicate introduced during the previous specialization is unfolded only once at the first step of the Unfolding phase. This choice guarantees the termination of the Unfolding phase (notice that new predicates can depend on themselves), and also avoids an undesirable, excessive increase of the number of clauses in the specialized program. Obviously, more sophisticated strategies for guiding unfolding could be applied. For instance, one may allow unfolding only if it generates at most one clause. This unfolding policy, called *determinate unfolding* in [65], does not increase the number of clauses and is useful in most cases. However, as we will show in Section 6.4, our simple choice is effective in practice.

In order for $\mathit{Specialize}_{prop}$ to be effective, we need to use a generalization strategies that retains as much information as possible, while guaranteeing the termination of the specialization.

Now we will consider four generalization strategies and in Section 6.4 we will compare them with respect to their strength and efficacy for the verification of program properties. These generalization strategies are based on the widening and convex hull generalization operators presented in Chapter 4.

Let us first present two *monovariant* generalization strategies that during the construction of *Defs*, for any given atom `Q(X)` to be folded, introduce a new definition of the form `newp(X) :- c(X), Q(X)`, which is more general than any other definition in *Defs* with the atom `Q(X)` in its body. Thus, when using these generalization operators, the definitions in *Defs* whose body contains the atom `Q(X)` are linearly ordered with respect to the 'more general' relation.

Monovariant Generalization with Widening. This strategy is denoted Gen_M. Let E be a clause of the form `H(X) :- e(X,X1), Q(X1)` and *Defs* be a set of predicate definitions. Then,
($\mu 1$) if in *Defs* there is no clause whose body atom is (a variant of) `Q(X1)`, then $\mathit{Gen}_M(E, \mathit{Defs})$ is defined as the clause `newp(X1) :-` $\mathtt{e}_p$`(X1), Q(X1)`, and
($\mu 2$) if in *Defs* there is a definition D of the form `newq(X1) :- d(X1), Q(X1)` and D is the most general such definition in *Defs*, then $\mathit{Gen}_M(E, \mathit{Defs})$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to $\mathtt{e}_p$`(X1)`.

Note that at any given time the last definition of the form: `newp(X1) :- c(X1), Q(X1)` that has been introduced in *Defs* is the most general one with the atom `Q(X1)` in its body.

Monovariant Generalization with Widening and Convex Hull. This strategy, denoted Gen_{MH}, alternates the computation of convex hull and widening. Indeed, in Case ($\mu 2$) above, if D has been derived by projection or widening, then $\mathit{Gen}_{MH}(E, \mathit{Defs})$ is `newp(X1) :- ch(X1), Q(X1)`, where `ch(X1)` is the convex hull of `d(X1)` and $\mathtt{e}_p$`(X1)`. In all other cases the function $\mathit{Gen}_{MH}(E, \mathit{Defs})$ is defined like the function $\mathit{Gen}_M(E, \mathit{Defs})$.

Other generalization strategies can be defined by computing any fixed number of consecutive convex hulls before applying widening.

Now we present two *polyvariant* generalization strategies, which may introduce several distinct, specialized definitions (with different constraints) for each atom. Polyvariant strategies allow, in principle, more precision with respect to monovariant operators, but in some cases they could also cause the introduction of too many new predicates, and hence an increase of both the size of the specialized program and the time needed for verification. We will consider this issue in Section 6.4 when we discuss the outcome of our experiments.

When we use a polyvariant generalization strategies, for any given atom `Q(X)`, the definitions in *Defs* whose body contains `Q(X)` are not necessarily linearly ordered with respect to the 'more general' relation. However, the definitions in the same path of the definition tree *Defs* are linearly ordered, and the last definition introduced is more general than all its ancestors.

Polyvariant Generalization with Widening. This strategy is denoted Gen_P. Let E be a clause of the form `H(X) :- e(X,X1), Q(X1)` and *Defs* be a *tree* of predicate definitions. Suppose that E has been derived by unfolding a definition C. Then, (π1) if in *Defs* there is no ancestor of C whose body atom is of the form `Q(X1)`, then $Gen_P(E, Defs)$ is the clause `newp(X1) :-` $\mathtt{e}_p$`(X1), Q(X1)`, and
(π2) if C has a most recent ancestor D in *Defs* of the form: `newq(X1) :- d(X1), Q(X1)`, then $Gen_P(E, Defs)$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to $\mathtt{e}_p$`(X1)`.

Polyvariant Generalization with Widening and Convex Hull. This strategy, denoted Gen_{PH}, is defined as Gen_P, except that it alternates the computation of convex hull and widening. Formally, Case (π2) above is modified as follows:
($\pi 2_{PH}$) if C has a most recent ancestor D in *Defs* of the form `newq(X1) :- d(X1), Q(X1)` that has been derived by projection or widening, then $Gen_{PH}(E, Defs)$ is the clause `newp(X1) :- ch(X1), Q(X1)`, where `ch(X1)` is the convex hull of `d(X1)` and $\mathtt{e}_p$`(X1)`, else $Gen_{PH}(E, Defs)$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to $\mathtt{e}_p$`(X1)`.

Now we show that all four strategies guarantee the soundness and termination of $Specialize_{prop}$.

Proposition 2. The strategy Gen_M, Gen_{MH}, Gen_P, and Gen_{PH} are generalization strategies.

Proof. By Definition 8, we have to show that, for each strategy *Gen* in {Gen_M, Gen_{MH}, Gen_P, and Gen_{PH}}, the following two properties hold.
Property (P1): for every clause E of the form: `H(X) :- e(X,X1), Q(X1)`, for every clause `newp(X1) :- g(X1), Q(X1)` obtained by applying the strategy *Gen* to E and some set *Defs* of definitions, we have that `e(X,X1)` $\sqsubseteq$ `g(X1)`, and

Property (P2): for every infinite sequence $E_1, E_2, \ldots$ of clauses, for every infinite sequence $G_1, G_2, \ldots$ of clauses constructed as follows: (1) $G_1 = Gen(E_1, \emptyset)$, and (2) for every $i>0$, $G_{i+1} = Gen(E_{i+1}, \{G_1, \ldots, G_i\})$, there exists an index k such that, for every $i>k$, G_i is equal, modulo the head predicate name, to a clause in $\{G_1, \ldots, G_k\}$.

(Gen_M is a generalization strategy)
– Let us prove that Property (P1) holds for Gen_M. If `g(X1)` is $\mathtt{e}_p$`(X1)` (see case (μ1) above), then `e(X,X1)` $\sqsubseteq$ $\mathtt{e}_p$`(X1)` because $\mathtt{e}_p$`(X1)` is the projection of `e(X,X1)` onto `X1`. If `g(X1)` is `w(X1)` (see case (μ2) above), then `e(X,X1)` $\sqsubseteq$ `w(X1)` because `w(X1)` is the widening of `d(X1)` with respect to $\mathtt{e}_p$`(X1)`, and hence $\mathtt{e}_p$`(X1)` $\sqsubseteq$ `w(X1)`. Thus, `e(X,X1)` $\sqsubseteq$ `g(X1)`.
– Let us prove that Property (P2) holds for Gen_M. This property is a straightforward consequence of the following two facts:

(i) each new definition introduced by Gen_M is a clause of the form: `newp(X1) :- g(X1),Q(X1)` , where `Q(X1)` is a function-free atom whose predicate symbol occurs in the input program I (recall that program I is obtained by $Specialize_{vcg}$, and this strategy removes, by folding, all function symbols occurring in the atoms of its input program); hence case (μ1) can occur a finite number of times only, and

(ii) if `g(X1)` is the widening of `d(X1)` with respect to $\mathtt{e}_p$`(X1)` and `g(X1)` is different from `d(X1)`, then the set of atomic constraints of `g(X1)` is a proper subset of the atomic constraints of `d(X1)`, and hence case (μ2) will eventually generate new predicate definitions whose body is equal to the body of previously generated definitions.

(Gen_{MH} is a generalization strategy)
The proof is similar to that for Gen_M.
– In order to prove that Property (P1) holds for Gen_{MH}, we use the fact that `e(X,X1)` $\sqsubseteq$ `ch(X1)`. Indeed, `e(X,X1)` $\sqsubseteq$ $\mathtt{e}_p$`(X1)` (because $\mathtt{e}_p$`(X1)` is the projection of `e(X,X1)` onto `X1`) and $\mathtt{e}_p$`(X1)` $\sqsubseteq$ `ch(X1)` (because `ch(X1)` is the convex hull of `d(X1)` and $\mathtt{e}_p$`(X1)`).
– In order to show that Property (P2) holds for Gen_{MH}, it is enough to note that Property (P2) is preserved if one interleaves the projection and widening operators with an application of the convex hull operator, and hence the proof already done for Gen_M readily extends to Gen_{MH}.

(Gen_P and Gen_{PH} are generalization strategies)
The proof is a straightforward extension of the proof for Gen_M and Gen_{MH}, respectively. In particular, in order to show that Property (P2) holds for Gen_P

and Gen_{PH}, it suffices to use the following fact. Suppose that $G_1, G_2, \ldots$ is an infinite sequence of predicate definitions. Let T be an infinite *tree* of definitions such that:
(i) if G occurs in $G_1, G_2, \ldots$, then G occurs in T,
(ii) if A is an ancestor of B in T, then A precedes B in $G_1, G_2, \ldots$ (that is, $G_1, G_2, \ldots$ is a linear order consistent with the ancestor relation in T),
(iii) T is finitely branching, and
(iv) every branch in T stabilizes.
Then $G_1, G_2, \ldots$ stabilizes. □

Different strategies can be adopted for applying the folding rule during the FOLDING phase. These folding strategies depend on the generalization strategy that is used for introducing new definitions. In the case where we use a monovariant strategy (either Gen_M or Gen_{MH}), we fold every clause of the form `H(X) :- e(X,X1), Q(X1)` using the most general definition of the form `newq(X1) :- g(X1), Q(X1)` occurring in the set *Defs* obtained at the end of the execution of the *while-loop* (α). We call this strategy the *most general folding strategy.* In the case where we use a polyvariant strategy (that is, Gen_P or Gen_{PH}), we fold every clause E using the definition computed by applying the generalization strategy to E. We call this strategy the *immediate folding strategy.*

Example 6 (Propagation of the constraints of the initial configuration). Let us consider the program in Example 2. We perform our second program specialization starting from the CLP program `V` we have derived by applying the Verification Conditions Generation step in Example 4 on page 47. This second specialization propagates the constraint '`X=0, Y=0`' characterizing the initial configuration which occurs in clause 32.

We apply the *Specialize*$_{prop}$ strategy with the Gen_P generalization operator. We start off by executing the *while-loop* (α) that repeats the UNFOLDING, CLAUSE REMOVAL, and DEFINITION INTRODUCTION phases.

First execution of the body of the while-loop (α).
UNFOLDING. We unfold clause 32 with respect to the atom `new1(X,Y,N)` and we get:

```
35. incorrect :- X1=1, Y1=1, N>0, new1(X1,Y1,N).
```

(Note that `new1(X,Y,N)` is also unifiable with the head of clause 34, but the constraint '`X=0,Y=0,X≥N,X>Y`' is unsatisfiable.) No CLAUSE REMOVAL can be applied.

Definition Introduction. Clause 35 cannot be folded, and hence we define the following new predicate:

36. `new2(X,Y,N) :- X=1, Y=1, N>0, new1(X,Y,N).`

Thus, the set *Defs* of new predicate definitions consists of clause 36 only.

Second execution of the body of the while-loop (α).
Unfolding. Now we unfold the last definition which has been introduced (clause 36) and we get:

37. `new2(X,Y,N) :- X=1, Y=1, X1=2, Y1=3, N>1, new1(X1,Y1,N).`

Definition Introduction. Clause 37 cannot be folded by using any definition in *Defs*. Thus, we apply a generalization operator Gen_P based on widening. This operator matches the constraint appearing in the body of clause 37 against the constraint appearing in the body of clause 36, which is the only clause in *Defs*. First, the constraint in clause 36 is rewritten as a conjunction `c` of inequalities: `X≥1,X≤1,Y≥1,Y≤1,N>0`. Then the variables of clause 37 are renamed so that the atom in its body is identical to the atom in the body of clause 36, as follows:

38. `new2(Xr,Yr,Nr) :- Xr=1, Yr=1, X=2, Y=3, N>1, new1(X,Y,N).`

Then the generalization operator Gen_P computes the projection of the constraint appearing in clause 38 onto the variables of the atom `new1(X,Y,N)`, which is the constraint `d`: `X=2,Y=3,N>1`. The widening of `c` with respect to `d` is the constraint '`X≥1,Y≥1,N>0`' obtained by taking the atomic constraints of `c` that are entailed by `d`. Thus, Gen_P introduces the following new predicate definition:

39. `new3(X,Y,N) :- X≥1, Y≥1, N>0, new1(X,Y,N).`

which is added to *Defs*.

Third execution of the body of the while-loop (α).
Unfolding. We unfold clause 39 and we get:

40. `new3(X,Y,N) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).`
41. `new3(X,Y,N) :- X≥N, X>Y, Y≥1, N>0.`

Clause 40 can be folded using clause 40 in *Defs*. Thus, the while-loop (α) terminates without introducing any new definition.

Folding. Now we fold clause 35 using definition 36 and clauses 37 and 40 using definition 39. We get the following final program `S`:

42. `incorrect :- N>0, X1=1, Y1=1, new2(X1,Y1,N).`
43. `new2(X,Y,N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1,Y1,N).`
44. `new3(X,Y,N) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y, new3(X1,Y1,N).`

45. `new3(X,Y,N) :- X≥N, X>Y, Y≥1, N>0.`

The application of *Specialize*$_{prop}$ has propagated the constraints defining the initial configuration. For instance, the constrained fact (clause 45) has the extra constraint '`Y≥1, N>0`', with respect to the constrained fact in program `V` which has only the constraint '`X≥N, X>Y`' (see clause 34 on page 50). However, in this example the presence of a constrained fact does not allow us to conclude that it has an empty least model, and hence at this point we are not able show the correctness of our program `sum`. □

5.2.2 Lightweight Correctness Analysis

The procedure *CorrectnessTest* (see Figure 9) analyzes the CLP program `S` and tries to determine whether or not `incorrect` belongs to M(`S`).

The analysis performed by *CorrectnessTest* is lightweight in the sense that, unlike the *Iterated Specialization* strategy, it always terminates, possibly returning the answer '*unknown*'.

Note that the output `S` of *Specialize*$_{prop}$ is a linear program. The *CorrectnessTest* procedure transforms `S` into a new, linear CLP program `Q` by using two auxiliary functions: (1) the *UnfoldCfacts* function, which takes a linear CLP program $\mathtt{Q}_1$ and replaces as long as possible a clause C in $\mathtt{Q}_1$ by $\mathit{Unf}(C, \mathtt{A})$, whenever `A` is defined by a set of constrained facts, and (2) the *Remove* function, which takes a linear CLP program $\mathtt{Q}_2$ and removes every clause C that satisfies one of the following two conditions: *either* (i) the head predicate of C is useless in $\mathtt{Q}_2$, *or* (ii) C is subsumed by a clause in $\mathtt{Q}_2$ distinct from C.

The *CorrectnessTest* procedure iterates the application of the function *UnfoldCfacts* followed by *Remove* until a fixpoint, say `Q`, is reached.

Now we prove that *CorrectnessTest* constructs program `Q` in a finite number of steps. Moreover, `Q` is equivalent to `S` with respect to the least model semantics, and hence `incorrect` $\in M$(`S`) iff `incorrect` $\in M$(`Q`).

Theorem 5 (Termination and Correctness of *CorrectnessTest*). Let `S` be a linear CLP program defining predicate `incorrect`. Then, (i) *CorrectnessTest* terminates for the input program `S`, and (ii.1) if *CorrectnessTest* (`S`) = '*correct*' then `incorrect` $\not\in M$(`S`), and (ii.2) if *CorrectnessTest* (`S`) = '*incorrect*' then `incorrect` $\in M$(`S`).

Proof. (i) the procedure *CorrectnessTest* constructs a fixpoint of the function $\lambda\mathtt{Q}.\mathit{Remove}(\mathit{UnfoldCfacts}(\mathtt{Q}))$ in a finite number of steps, as we now show. For a CLP program P, let $\mathit{pn}(P)$ denote the number of distinct predicate symbols

Input: A linear CLP program `S` defining the predicate `incorrect`.
Output: Either '*correct*' (implying $\texttt{incorrect} \notin M(\texttt{S})$), or '*incorrect*' (implying $\texttt{incorrect} \in M(\texttt{S})$), or '*unknown*'.

```
Q := S;
while  Q ≠ Remove(UnfoldCfacts(Q))  do
    Q := Remove(UnfoldCfacts(Q));
end-while;
if      incorrect has a fact in Q  then  return 'incorrect'
elseif  no clause in T has predicate  incorrect  then  return 'correct'
else    return 'unknown'
```

Figure 9: The *CorrectnessTest* Procedure.

that occur in the body of a clause in P, and let $cn(P)$ denote the number of clauses in P. Then, the following facts hold:
(1) either $P = \mathit{UnfoldCfacts}(P)$ or $pn(P) > pn(\mathit{UnfoldCfacts}(P))$,
(2) $pn(P) \geq pn(\mathit{Remove}(P))$,
(3) either $P = \mathit{Remove}(P)$ or $cn(P) > cn(\mathit{Remove}(P))$.
Thus, we have that $\langle pn(P), cn(P)\rangle \geq_{lex} \langle pn(\mathit{Remove}(\mathit{UnfoldCfacts}(P))), cn(\mathit{Remove}(\mathit{UnfoldCfacts}(P)))\rangle$, where $\geq_{lex}$ is the lexicographic ordering on pairs of integers.

Since $>_{lex}$ is well-founded, *CorrectnessTest* eventually gets a program `Q` s.t. $\langle pn(\texttt{Q}), cn(\texttt{Q})\rangle = \langle pn(\mathit{Remove}(\mathit{UnfoldCfacts}(\texttt{Q}))), cn(\mathit{Remove}(\mathit{UnfoldCfacts}(\texttt{Q})))\rangle$, and hence, by (1) and (3), $\texttt{Q} = \mathit{Remove}(\mathit{UnfoldCfacts}(\texttt{Q}))$.
(ii.1) No application of the folding rule is performed by *CorrectnessTest*, and hence the condition for the correctness of the transformation rules of Theorem 1 is trivially satisfied. Thus, $\texttt{incorrect} \in M(\texttt{S})$ iff $\texttt{incorrect} \in M(\texttt{Q})$. If *CorrectnessTest* $(\texttt{S}) =$ '*correct*', then no clause in `Q` has predicate `incorrect`, and then $\texttt{incorrect} \notin M(\texttt{Q})$. Hence, $\texttt{incorrect} \notin M(\texttt{S})$.
(ii.2) If *CorrectnessTest*$(\texttt{S}) =$ '*incorrect*', then a fact in `Q` has predicate `incorrect`, and $\texttt{incorrect} \in M(\texttt{Q})$. Since at Point (ii) we have shown that $\texttt{incorrect} \in M(\texttt{S})$ iff $\texttt{incorrect} \in M(\texttt{Q})$, we conclude that $\texttt{incorrect} \in M(\texttt{S})$. □

In our running example, program `S` is the CLP program obtained by applying the procedure $\mathit{Specialize}_{prop}$. Now program `S` consists of clauses 42–45 and therefore we are not able to determine whether or not the atom `incorrect` is a consequence of `S`. Indeed, (i) in program `S` no predicate is defined by constrained facts only, and hence *UnfoldCfacts* has no effect, (ii) in `S` no predicate is useless and no clause is subsumed by any other, and hence also *Remove* leaves `S` unchanged, and (iii) in `S` there is a clause for `incorrect` which is not a fact.

5.2.3 The Reverse Transformation

The *Reverse* procedure implements a transformation that reverses the flow of computation: the top-down evaluation (that is, the evaluation from the head to the body of a clause) of the transformed program corresponds to the bottom-up evaluation (that is, the evaluation from the body to the head) of the given program. In particular, if the *Reverse* procedure is applied to a program that checks the reachability of the error configurations by exploring the transitions in a forward way starting from the initial configurations, then the reversed program checks reachability of the initial configurations by exploring the transitions in a backward way starting from error configurations. Symmetrically, from a program that checks reachability by a backward exploration of the transitions, *Reverse* derives a program that checks reachability by a forward exploration of the transitions.

Let us consider a linear CLP program `S` of the form:

$$
\begin{array}{l}
\texttt{incorrect :- } \texttt{a}_1\texttt{(X), p}_1\texttt{(X).}\\
\quad \ldots\\
\texttt{incorrect :- } \texttt{a}_k\texttt{(X), p}_k\texttt{(X).}\\
\texttt{q}_1\texttt{(X) :- t}_1\texttt{(X,X1), r}_1\texttt{(X1).}\\
\quad \ldots\\
\texttt{q}_m\texttt{(X) :- t}_m\texttt{(X,X1), r}_m\texttt{(X1).}\\
\texttt{s}_1\texttt{(X) :- b}_1\texttt{(X).}\\
\quad \ldots\\
\texttt{s}_n\texttt{(X) :- b}_n\texttt{(X).}
\end{array}
$$

where: (i) $\texttt{a}_1\texttt{(X)}, \ldots, \texttt{a}_k\texttt{(X)}, \texttt{t}_1\texttt{(X,X1)}, \ldots, \texttt{t}_m\texttt{(X,X1)}, \texttt{b}_1\texttt{(X)}, \ldots, \texttt{b}_n\texttt{(X)}$ are constraints, and (ii) the $\texttt{p}_i$'s, $\texttt{q}_i$'s, $\texttt{r}_i$'s, and $\texttt{s}_i$'s are possibly non-distinct predicate symbols.

The *Reverse* procedure transforms program `S` in two steps as follows.

Step 1. Program `S` is transformed into a program `T` of the following form (round parentheses make a single argument out of a tuple of arguments):

$$
\begin{array}{ll}
\text{t1.} & \texttt{incorrect :- a(U), r1(U).}\\
\text{t2.} & \texttt{r1(U) :- trans(U,V), r1(V).}\\
\text{t3.} & \texttt{r1(U) :- b(U).}\\
 & \texttt{a((p}_1\texttt{,X)) :- a}_1\texttt{(X).}\\
 & \ldots
\end{array}
$$

```
a((p$_k$,X)) :- a$_k$(X).
trans((q$_1$,X),(r$_1$,X1)) :- t$_1$(X,X1).
 ...
trans((q$_m$,X),(r$_m$,X1)) :- t$_m$(X,X1).
b((s$_1$,X)) :- b$_1$(X).
 ...
b((s$_n$,X)) :- b$_n$(X).
```

Step 2. Program `T` is transformed into a program `R` by replacing the first three clauses t1–t3 of `T` with the following ones:

r1. `incorrect :- b(U), r2(U).`
r2. `r2(V) :- trans(U,V), r2(U).`
r3. `r2(U) :- a(U).`

The correctness of the transformation of `S` into `R` is shown by the following result.

Theorem 6 (Soundness of the Reverse procedure). Let `R` be the program derived from program `S` by the *Reverse* procedure. Then `incorrect`$\in M$(`R`) iff `incorrect`$\in M$(`S`).

Proof. (*Step* 1.) By unfolding clauses t1, t2, and t3 of program `T` with respect to `a(U)`, `trans(U,V)`, and `b(U)`, respectively, we get the following CLP program `T1`:

```
incorrect :- a$_1$(X), r1((p$_1$,X)).
    ...
incorrect :- a$_k$(X), r1((p$_k$,X)).
r1((q$_1$,X)) :- t$_1$(X,X1), r1((r$_1$,X1)).
    ...
r1((q$_m$,X)) :- t$_m$(X,X1), r1((r$_m$,X1)).
r1((s$_1$,X)) :- b$_1$(X).
    ...
r1((s$_n$,X)) :- b$_n$(X).
```

By the correctness of the unfolding rule (see Theorem 1), we get that `incorrect`$\in M$(`T`) iff `incorrect`$\in M$(`T1`).

Then, by rewriting all atoms in `T1` of the form `r1((pred,Z))` into `pred(Z)`, we get back `R`. (The occurrences of predicate symbols in the arguments of `a`, `trans`, and `b` should be considered as individual constants.) The correctness of this rewriting is straightforward, as it is based on the syntactic isomorphism between `S` and `T1`. A formal proof of correctness can be made by observing that the above rewriting can also realized by introducing predicate definitions of the form `pred(Z) :- r1((pred,Z))`, and applying the unfolding and folding rules. Thus, `incorrect` $\in M($`T1`$)$ iff `incorrect` $\in M($`S`$)$.

(*Step* 2.) The transformation of `T` into `R` (and the opposite transformation) can be viewed as a special case of the *grammar-related transformation* studied in [23]. We refer to that paper for a proof of correctness. Thus, we have that `incorrect` $\in M($`T`$)$ iff `incorrect` $\in M($`R`$)$, and we get the thesis. □

The predicates `a`, `trans`, and `b` are assumed to be unfoldable in the subsequent application of the *Specialize*$_{prop}$ procedure.

Example 7 (Propagation of the constraints of the error configuration). Now let us continue our running example. The program `S` of Example 6 derived by the *Specialize*$_{prop}$ procedure can be transformed into a program `R` of the form t1–t3, where the predicates `a`, `trans`, and `b` are defined as follows:

```
45. a((new2,X1,Y1,N)) :- N>0, X1=1, Y1=1.
46. trans((new2,X,Y,N),(new3,X1,Y1,N)) :- X=1, Y=1, N>1, X1=2, Y1=3.
47. trans((new3,X,Y,N),(new3,X1,Y1,N)) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y.
48. b((new3,X,Y,N)) :- Y≥1, N>0, X≥N, X>Y.
```

Then, the reversed program `R` is as follows.

```
49. incorrect :- b(U), r2(U).
50. r2(V) :- trans(U,V), r2(U).
51. r2(U) :- a(U).
```

together with clauses 45–48 above.

The idea behind program reversal is best understood by considering the reachability relation in the (possibly infinite) transition graph whose transitions are defined by the (instances of) clauses 46 and 47. Program `S` checks the reachability of a configuration $c2$ satisfying `b(U)` from a configuration $c1$ satisfying `a(U)`, by moving *forward* from $c1$ to $c2$. Program `R` checks the reachability of $c2$ from $c1$, by moving *backward* from $c2$ to $c1$. Thus, in the case where `a(U)` and `b(U)` are predicates that characterize the initial and final configurations, respectively, by the reversal transformation we derive a program that checks the reachability of an error configuration starting from an initial configuration by moving *backward* from the error configuration. In particular, in the body of the clause for `incorrect` in `R` the constraint `b(U)` contains, among others, the constraint `X>Y` characterizing the error configuration (see clause 48) and, by specializing `R`, we will propagate the constraint of the error configuration.

Now let us specialize the CLP program `R` consisting of clauses 49–51 and 45–48 by applying again *Specialize*$_{prop}$. We start off from the first while-loop (α) of that procedure.

First execution of the body of the while-loop (α).
UNFOLDING. We unfold the clause for `incorrect` (clause 49) with respect to the leftmost atom `b(U)` and we get:

52. `incorrect :- Y≥1, N>0, X≥N, X>Y, r2((new3,X,Y,N)).`

DEFINITION INTRODUCTION. In order to fold clause 52 we introduce the definition:

53. `new4(X,Y,N) :- Y≥1, N>0, X≥N, X>Y, r2((new3,X,Y,N)).`

Second execution of the body of the while-loop (α).
UNFOLDING. Then we unfold clause 53 and we get:

54. `new4(X,Y,N) :- Y≥1, N>0, X≥N, X>Y, a((new3,X,Y,N)).`
55. `new4(X1,Y1,N) :- Y1≥1, N>0, X1≥N, X1>Y1,`
    `trans(U,(new3,X1,Y1,N)), r2(U).`

By unfolding, clause 54 is deleted because the head of lause 45 is not unifiable with `a((new3,X,Y,N))`.
By unfolding clause 55 with respect to `trans(U,(new3,X1,Y1,N))`, also this clause is deleted because unsatisfiable constraints are derived. Thus, no new definition is introduced, and the while-loop (α) terminates.

FOLDING. By folding clause 52 using definition 53 we get:

56. `incorrect :- Y≥1, N>0, X≥N, X>Y, new4(X,Y,N).`

and the final, specialized CLP program `S` consists of clause 56 only.

Now we apply the *CorrectnessTest* procedure to program `S`, which detects that `incorrect` is a useless predicate. Then, the *Iterated Specialization* terminates by reporting the correctness of the given imperative program `sum`. □

Thus, in this example we have seen that by iterating the specializations which propagate the constraints occurring in the initial configuration and in the error configuration we have been able to show the correctness of the given program.

It can be shown that, if we perform our specializations (using the same unfolding and generalization procedures) by taking into account only the constraints of the initial configuration *or* only the constraints of the error configuration, it is *not* possible to prove program correctness in our example. Thus, as we advocate here, if we perform a sequence of program specializations, we may gain an extra power when we have to prove program properties. This is confirmed by the experiments we have performed on various examples taken from the literature. We will report on those experiments in Section 6.4.

5.2.4 Soundness of the Iterated Specialization Strategy

Finally, we get the following soundness result for the Iterated Specialization method.

Theorem 7 (Soundness of the Iterated Verification method). Let P be the CLP Encoding of the incorrectness problem for $\{\!\{\varphi_{init}\}\!\}$ Prog $\{\!\{\varphi_{error}\}\!\}$. If the Iterated Specialization strategy terminates for the input program P and returns '*correct*', then Prog is correct with respect to φ_{init} and φ_{error}. If the strategy terminates and returns '*incorrect*', then P is incorrect with respect to φ_{init} and φ_{error}.

Proof. The Iterated Specialization method terminates for the input program I and returns '*correct*' (respectively, '*incorrect*') if and only if there exists n such that:

$\mathtt{V}{:=}\mathit{Specialize}_{vcg}\,(\mathtt{P})$; $\mathtt{S}_1{:=}\mathit{Specialize}_{prop}(\mathtt{V})$;
$\mathtt{R}_2{:=}\mathit{Reverse}(\mathtt{S}_1)$; $\mathtt{S}_2{:=}\mathit{Specialize}_{prop}(\mathtt{R}_2)$;
...
$\mathtt{R}_n{:=}\mathit{Reverse}(\mathtt{S}_{n-1})$; $\mathtt{S}_n{:=}\mathit{Specialize}_{prop}(\mathtt{R}_n)$;

and $\mathit{CorrectnessTest}\,(\mathtt{S}_n) =$ '*correct*'
(respectively, $\mathit{CorrectnessTest}\,(\mathtt{S}_n) =$ '*incorrect*').
Then (by Theorem 5),
$\mathtt{incorrect} \notin M(\mathtt{S}_n)$ (respectively, $\mathtt{incorrect} \in M(\mathtt{S}_n)$)
if and only if (by Theorems 4 and 6 and Proposition 2)
$\mathtt{incorrect} \notin M(\mathtt{P})$ (respectively, $\mathtt{incorrect} \in M(\mathtt{P})$)
if and only if (by Theorem 2)
Prog is correct (respectively, incorrect) with respect to φ_{init} and φ_{error}. □

5.3 Experimental Evaluation

We have performed an experimental evaluation of our software model checking method on several benchmark programs taken from the literature. The results of our experiments show that our approach is competitive with state-of-the-art software model checkers.

The benchmark set used in our experiments consists of 216 verification problems of C programs (179 of which are correct, and the remaining 37 are incorrect). Most problems have been taken from the benchmark sets of other tools used in software model checking, like DAGGER [75] (21 problems), TRACER [85] (66 problems) and InvGen [76] (68 problems), and from the TACAS 2013 Software Verification Competition [12] (52 problems). The size of the input programs ranges from a dozen to about five hundred lines of code.

The verification problems came in different source formats (and, in particular, they used different methods for specifying the initial and error conditions), and thus they could not be directly used by other software model checkers. We automatically converted all problems from the original format to a common, intermediate format, and then from the intermediate format to the format accepted by each tool we have considered in our experiments. We have put great care and effort in the coding of the conversion programs to ensure maximum compatibility. Nonetheless, some software model checkers failed to run on some seemingly harmless verification problems. The source code of all the verification problems we have considered and detailed reports about the verification results are available at `http://map.uniroma2.it/VeriMAP/scp/`.

We have realized the VeriMAP software model checker [138] that implements our verification method (see Chapter 8). Our software model checker has been configured to execute the following program transformation:

$Specialize_{vcg}$; $Specialize_{prop}$; $CorrectnessTest$;
    ($Reverse$; $Specialize_{prop}$; $CorrectnessTest$)*

It executes: (i) a first program specialization consisting of a single application of the $Specialize_{vcg}$ procedure that performs the *removal of the interpreter*, and (ii) a sequence of applications of the $Specialize_{prop}$ procedure (from now on called *iterations*) that performs the propagation of the constraints of the initial and the error configurations. After the removal of the interpreter, the first application of $Specialize_{prop}$ propagates the constraints of the initial configuration. This corresponds to a *forward* propagation along the graph of configurations associated with the reachability relation `tr` (see Section 3.2), while the propagation of the constraints of the error configuration corresponds to a *backward* propagation along the graph of configurations. The $Specialize_{prop}$ procedure has been executed by using the four generalization operators presented in Section 5.1: (i) Gen_M, that is a monovariant generalization with widening only, (ii) Gen_{MH}, that is a monovariant generalization with widening and convex hull, (iii) Gen_P, that is a polyvariant generalization with widening only, and (iv) Gen_{PH}, that is polyvariant generalization with widening and convex hull.

We have also tested the following three state-of-the-art CLP-based software model checkers for C programs: ARMC [123], HSF(C) [73], and TRACER [85]. ARMC and HSF(C) are based on the Counter-Example Guided Abstraction Refinement technique (CEGAR) [29, 90, 128], while TRACER uses a technique based on approximated preconditions and approximated postconditions. We have compared the performance of those software model checkers with the performance of VeriMAP on our benchmark programs.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system Ubuntu 12.10 (64 bit) (kernel version 3.2.0-27). A timeout limit of five minutes has been set for all model checkers.

In Table 5.1 we summarize the verification results obtained by the four software verification tools we have considered. In the column labelled by VeriMAP (Gen_{PH}) we have reported the results obtained by using the VeriMAP system that implements our Iterated Specialization method with the generalization operator Gen_{PH}. In the remaining columns we have reported the results obtained, respectively, by ARMC, HSF(C), and TRACER using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options.

Line 1 reports the total number of correct answers of which those for correct problems and incorrect problems are indicated in line 2 and 3, respectively. Line 4 reports the number of verification tasks that ended with an incorrect answer. These verification tasks refer to correct programs that have been proved incorrect (*false alarms*, at line 5), and incorrect programs that have been proved correct (*missed bugs*, at line 6). Line 7 reports the number of verification tasks that aborted due to some errors (originating from inability of parsing or insufficient memory). Line 8 reports the number of verification tasks that did not provide an answer within the timeout limit of five minutes.

The total score obtained by each tool using the score function of the TACAS 2013 Software Verification Competition [12], is reported at line 9 and will be used in Figure 10. The score function assigns to every program p the integer *score*(p) determined as follows: (i) 2, if p is correct and has been correctly verified, (ii) 1, if p is incorrect and has been correctly verified, (iii) -4, if a false alarm has been generated, and (iv) -8, if a bug has been missed. Programs that caused errors or timed out do not contribute to the total score.

At line 9 we have indicated between round parentheses the negative component of the score due to false alarms and missed bugs. Line 10 reports the total CPU time, in seconds, taken to run the whole set of verification tasks: it includes the time taken to produce (correct or incorrect) answers and the time

		VeriMAP (Gen_{PH})	ARMC	HSF(C)	TRACER	
					SPost	*WPre*
1	*correct answers*	185	138	159	91	103
2	correct problems	154	112	137	74	85
3	incorrect problems	31	26	22	17	18
4	*incorrect answers*	0	9	5	13	14
5	false alarms	0	8	3	13	14
6	missed bugs	0	1	2	0	0
7	*errors*	0	18	0	20	22
8	*timed-out problems*	31	51	52	92	77
9	*total score*	339 (0)	210 (-40)	268 (-28)	113 (-52)	132 (-56)
10	*total time*	10717.34	15788.21	15770.33	27757.46	23259.19
11	*average time*	57.93	114.41	99.18	305.03	225.82

Table 5.1: Verification results using VeriMAP, ARMC, HSF(C) and TRACER. For each column the sum of the values of lines 1, 4, 7, and 8 is 216, which is the number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.

spent on tasks that timed out (we did not include the negligible time taken for tasks that aborted due to errors). Line 11 reports the average time needed to produce a correct answer, which is obtained by dividing the total time (line 10) by the number of correct answers (line 1).

On the set of verification problems we have considered, the VeriMAP system is able to provide correct answers to 185 problems out of 216. It is followed by HSF(C) (159 correct answers), ARMC (138), TRACER(*WPre*) (103), and TRACER(*SPost*) (91). Moreover, VeriMAP has produced no errors and no incorrect answers, while the other tools generate from 4 to 14 incorrect answers. Thus, VeriMAP exhibits the best precision, defined as the ratio between the number of correct answers and the number of verification problems.

The total time taken by VeriMAP is smaller than the time taken by any of the other tools we have considered. This result is somewhat surprising, if we consider the generality of our approach and the fact that our system has not been specifically optimized for software model checking. This good performance of VeriMAP is due to: (i) the small number of tasks that timed out, and (ii) the fact that VeriMAP takes very little time on most programs, while it takes much more time on a few, complex programs (see Figure 10). In particular, VeriMAP is able to produce 169 answers taking at most 5 seconds each, and this is indeed a good performance if we compare it with HSF(C) (154 answers), ARMC (122), TRACER(*SPost*) (88) and TRACER(*WPre*) (101).

In order to ease the comparison of the performance of the software model checkers we have considered, we adopt a visualization technique using score-based quantile functions (see Figure 10), similar to that used by the TACAS 2013 Software Verification competition [12]. By using this technique, the performance of each tool is represented by a quantile function, which is a set of pairs (x, y) computed from the following set of pairs:

$$V = \{\, (p, t) \mid \text{the correct answer for the (correct or incorrect) program } p \text{ is produced in } t \text{ seconds} \,\}.$$

Given the set V, for each pair $(p, T) \in V$ we produce a pair (x, y) computed as follows:

(i) $x = XS + \sum_{(p,\,t) \in V \,\wedge\, t \leq T} \mathit{score}(p)$, where XS, called the *x-shift*, is the sum of all the negative scores due to incorrect answers (x is called the *accumulated score*), and

(ii) $y = YS + T$, where YS, called the *y-shift*, is a number of seconds equal to the number of the timed-out programs.

Quantile functions are discrete monotone functions, but for reasons of simplicity,

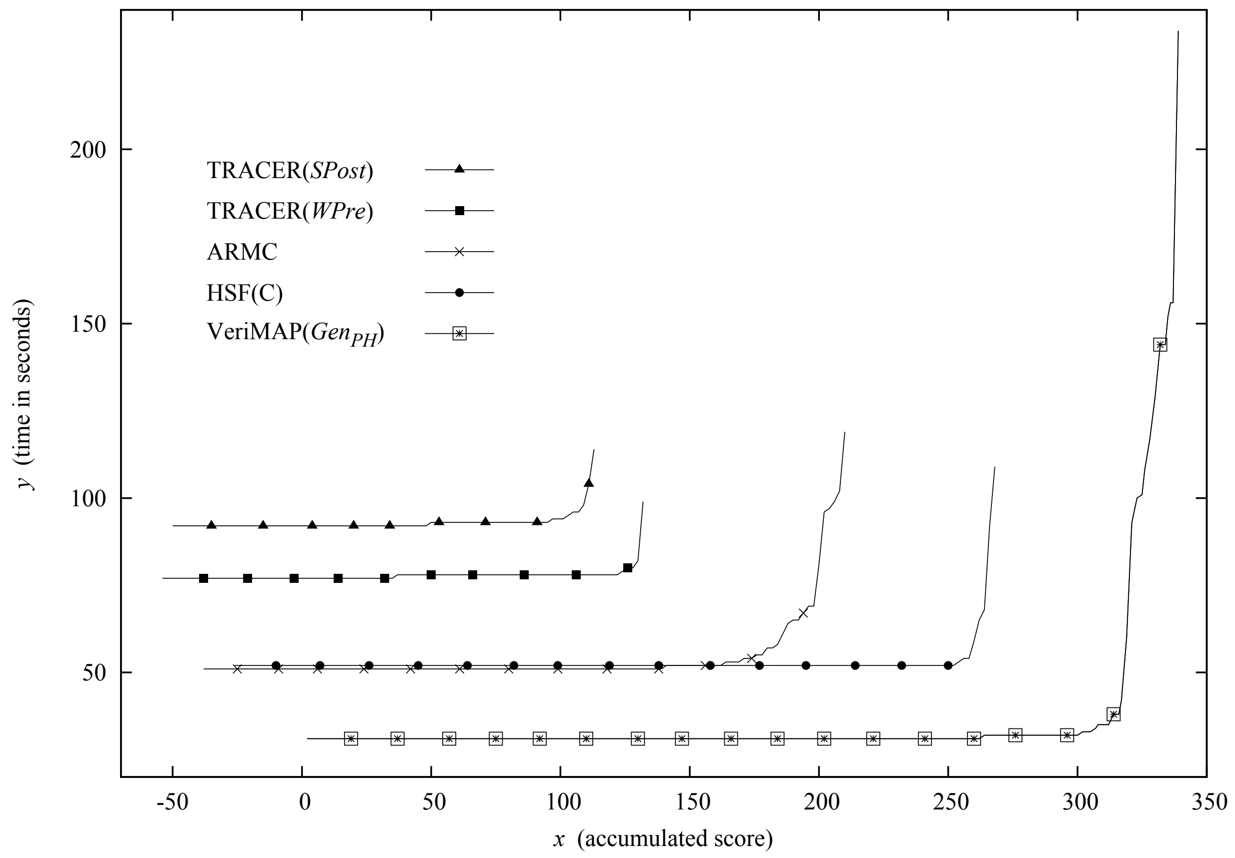


Figure 10: Score-based quantile functions for TRACER(*SPost*), TRACER(*WPre*), ARMC, HSF(C), and VeriMAP(Gen_{PH}). Markers are placed along the lines of the functions, from left to right, every 10 programs that produce correct answers, starting from the program whose verification took the minimum time.

in Figure 10 we depicted them as continuous lines. The line of a quantile function for a generic verification tool should be interpreted as follows:
(i) the x coordinate of the k-th leftmost point of the line, is the sum of the score of the fastest k correctly verified programs plus the (non-positive) *x-shift*, and measures the reliability of the tool,
(ii) the y coordinate of the k-th leftmost point of the line, is the verification time taken for the k-th fastest correctly verified program plus the (non-negative) *y-shift*, and measures the inability of the tool of providing an answer (either a correct or an incorrect one),
(iii) the span of the line along the x-axis, called the *x-width*, measures the precision of the tool, and
(iv) the span of the line along the y-axis, called the *y-width*, is the difference of verification time between the slowest and the fastest correctly verified programs.

Moreover, along each line of the quantile functions of Figure 10 we have put a marker every ten answers. In particular, for $n \geq 1$, the n-th marker, on the left-to–right order, denotes the point (x, y) corresponding to the $(10 \times n)$-th fastest correct answer that has been produced.

Informally, for the line of a quantile function we have that: good results move the line to the right (because of lower total negative score XS), stretch it horizontally (because of higher positive accumulated score), move it towards the x-axis (because of fewer timed-out problems and a better time performance), and compress it vertically (because of a lower difference between worst-case and best-case time performance). We observe that the line of the quantile function for VeriMAP(Gen_{PH}) starts with a positive x-value (indeed, VeriMAP provides no incorrect answers and $XS = 0$), is the widest one (indeed, VeriMAP has the highest positive accumulated score, due to its highest precision among the tools we have considered), and is the lowest one (indeed, VeriMAP(Gen_{PH}) has the smallest numbers of timed-out problems). The height of the line for VeriMAP(Gen_{PH}) increases only towards the right end (at an accumulated score value of 300) after having produced 162 correct answers, and this number is greater than the number of all the correct answers produced by any of the other tools we have considered.

In Table 5.2 we report the results obtained by running the VeriMAP system with the four generalization operators presented in Section 5.1.

Each column is labelled by the name of the associated generalization operator. Line 1 reports the total number of correct answers. Lines 2 and 3 report the number of correct answers for correct and incorrect problems, respectively. Line 4 reports the number of tasks that timed out. As already mentioned, the

VeriMAP system has produced neither incorrect answers nor errors. Line 5 reports the total time, including the time spent on tasks that timed out, taken to run the whole set of verification tasks. Line 6 reports the average time needed to produce a correct answer, that is, the total time (line 5) divided by the number of correct answers (line 1). Line 7 reports the total correct answer time, that is the total time taken for producing the correct answers, excluding the time spent on tasks that timed out. Lines 7.1–7.4 report the percentages of the total correct answer time taken to run the *C2CLP* module, the $\mathit{Specialize}_{vcg}$ procedure, the $\mathit{Specialize}_{prop}$ procedure, and the *CorrectnessTest* procedure, respectively. Line 8 reports the average correct answer time, that is, the total correct answer time (line 7) divided by the number of correct answers (line 1). Line 9 reports the maximum number of iterations of the Iterated Specialization strategy that were needed, after the removal of the interpreter, for verifying the correctness property of interest on the various programs. Line 10 reports the total number of predicate definitions introduced by the $\mathit{Specialize}_{prop}$ procedure during the sequence of iterations.

The data presented in Table 5.2 show that polyvariance always gives better precision than monovariance. Indeed, the polyvariant generalization operator with convex hull Gen_{PH} achieves the best precision (it provides the correct answer for 85.65% of 216 programs of our benchmark), followed by the polyvariant generalization operator without convex hull Gen_{P} (73.61%). (For this reason in Table 5.1 we have compared the other verification systems against VeriMAP (Gen_{PH}).) As already mentioned, polyvariant generalization may introduce more than one definition for each program point, which means that the *Specialize* procedure yields a more precise abstraction of the program to be verified and, consequently, it may increase the effectiveness of the analysis. The increase of precision obtained by using polyvariant operators rather than monovariant ones is particularly evident when proving incorrect programs (the precision is increased of about 100%, from 15 to 29–31).

On the other hand, monovariant operators enjoy the best trade-off between precision and average correct answer time. For example, when considering the average correct answer time, the Gen_{M} operator, despite the significant loss of precision (about 20%, from 159 to 128) with respect to its polyvariant counterpart Gen_{P}, is about 140% faster (3.25 seconds versus 7.88 seconds), when we consider the average correct answer time.

The good performance of monovariant operators is also justified by the much smaller number of definitions (about one tenth) introduced by the $\mathit{Specialize}_{prop}$ procedure with respect to those introduced in the case of polyvariant operators.

		VeriMAP generalization operators			
		Gen_{M}	Gen_{MH}	Gen_{P}	Gen_{PH}
1	*correct answers*	128	147	159	185
2	correct problems	113	132	130	154
3	incorrect problems	15	15	29	31
4	*timed-out problems*	88	69	57	31
5	*total time*	26816.64	21362.93	18353.11	10717.34
6	*average time*	209.51	145.33	115.43	57.93
7	*total correct answer time*	416.64	662.93	1253.11	1417.34
7.1	*C2CLP*	2.27%	1.63%	0.96%	0.98%
7.2	$Specialize_{Remove}$	6.81%	4.74%	4.44%	4.06%
7.3	$Specialize_{prop}$	90.33%	93.16%	42.77%	44.68%
7.4	*CorrectnessTest*	0.59%	0.46%	51.83%	50.27%
8	*average correct answer time*	3.25	4.51	7.88	7.66
9	*max number of iterations*	4	7	10	7
10	*number of definitions*	5623	6248	54977	58226

Table 5.2: Verification results using the VeriMAP system with different generalization operators. The sum of the values of lines 1 and 4 is 216, which is the number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.

Also the size of the programs produced by monovariant operators is much smaller with respect to those produced by polyvariant operators. (The size of the final programs obtained by specialization is not reported in Table 5.2, but it is approximately proportional to the number of definitions.) This also explains why the impact on the total correct answer time of the *CorrectnessTest* analysis is much lower for monovariant operators (less than 1%) than for polyvariant operators (about 50%).

The weight of the two preliminary phases (translation from C to CLP and removal of the interpreter) on the overall verification process is very limited. The execution times for the *C2CLP* module are very low (about 50 milliseconds per program) and have a low impact on the total correct answer time (at most 2.27%, as reported on line 7.1). The time taken by $\mathit{Specialize}_{\mathit{Remove}}$ for removing the interpreter ranges from a few tenths of milliseconds to about four seconds, for the most complex programs, and its impact on the total correct answer time is between 4% and 7% (see line 7.2).

In the set of problems we have considered, the higher average correct answer time of polyvariant operators does not prevent them from being more precise than monovariant operators. Indeed, by using polyvariant operators we get fewer timed-out problems with respect to those obtained by using monovariant operators, and thus for the verifications that use polyvariant operators we also get smaller total and average times (which take into account the number of timed-out problems).

We also observe that generalization operators using convex hull always give greater precision than their counterparts that do not use convex hull. This confirms the effectiveness of the convex hull operator, which may help infer relations among program variables, and may ease the discovery of useful program invariants.

Some of the programs are verified by the VeriMAP system during the first iteration after the removal of the interpreter by propagating the constraints of the initial configuration only. Nonetheless, as indicated in Figure 11, which shows the precision achieved by the VeriMAP generalization operators during the first ten iterations, our approach of iterating program specialization is effective and determines a significant increase of the number of correct answers (up to 115% for Gen_P, from 74 to 159). Moreover, we have that the verification process is able to prove programs by performing up to 7 or 10 iterations when using the more precise generalization operators Gen_{PH} or Gen_P, respectively.

The highest increase of precision is given by the second iteration and, although most correct answers are provided within the fourth iteration, all generalization

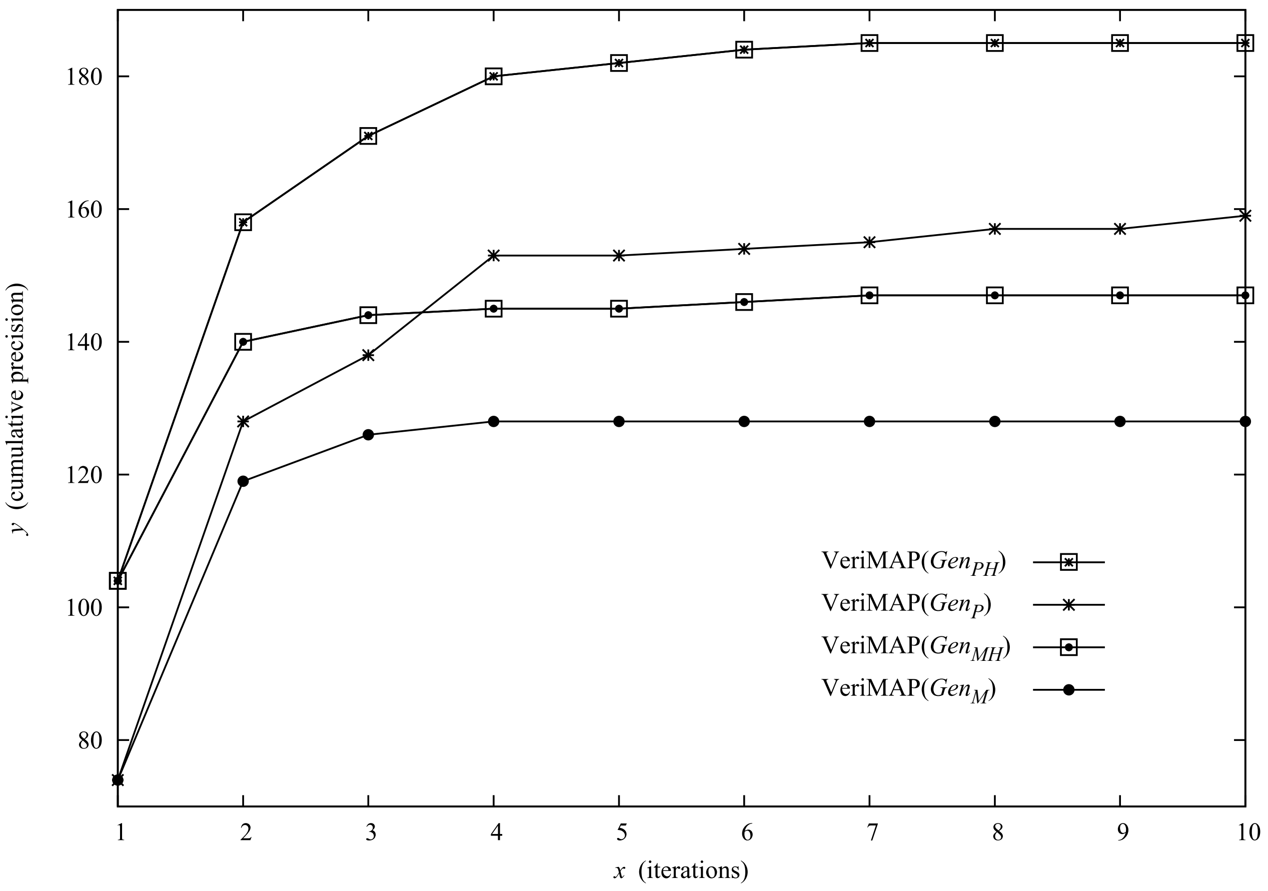


Figure 11: Cumulative precision achieved by the VeriMAP generalization operators during the first ten iterations.

operators (except for the monovariant operator Gen_M) keep increasing their precision from the fifth iteration onwards, providing up to 6 additional correct answers each.

The iteration of specialization is more beneficial when using polyvariant generalization operators where the increase of the number of answers reaches 115%, while the increase for monovariant generalization operators is at most 52%. For example, at the first iteration Gen_P is able to prove less properties than Gen_{MH}, but it outperforms the monovariant operator with convex hull by recovering precision when iterating the specialization.

The increase of precision due to the iteration of program specialization is also higher for generalization operators that do not use the convex hull operator (Gen_M and Gen_P), compared to their counterparts that use convex hull (Gen_{MH} and Gen_{PH}).

We would also like to point out that the use of convex hull is very useful during the first iterations. Indeed, the generalization operators using convex hull can verify 104 programs at the first iteration, while operators not using convex hull can verify 74 programs only. In this case the choice of using a polyvariant vs. monovariant generalization operator has no effect on the number of verified programs at the first iteration.

Finally, we note that the sets of programs with correct answers by using different operators are not always comparable. Indeed, the total number of different programs with correct answers by using any of the generalization operators we have considered is 190, while the programs for which a single operator produced a correct answer is at most 185. Moreover, there are programs that can be verified by operators with lower precision but that cannot be verified by operators with higher precision, within the considered timeout limit. For example, in our experiments the least precise operator Gen_M was able to prove four programs for which the most precise operator Gen_{PH} timed out.

This confirms that different generalization operators can give, in general, different results in terms of precision and performance. If we do not consider time limitations, generalization operators having higher precision should be preferred over less precise ones, because they may offer more opportunities to discover the invariants that are useful for proving the properties of interest. In some cases, however, the use of more precise generalization operators determines the introduction of more definition clauses, each requiring an additional iteration of the while-loop (α) of the *Specialize* Procedure, thereby slowing down the verification process and possibly preventing the termination within the considered timeout limit. In practice, the choice of the generalization operator to be used for any given verification problem at hand, can be made according to some heuristics that may be provided on the basis of the above mentioned trade-off between precision and efficiency.

5.4 Related Work

Our work is related to [16, 127], not only for the common use of constraints and Horn clauses, but also for the generality of the approach. Indeed, the technique we have presented here can be seen as a particular application of a more general verification method based on CLP and program transformation. CLP is used for specifying: (i) the programming language under consideration and its semantics, and (ii) the logic used for expressing the properties of interest and its proof rules, and CLP program transformation is used as a general-purpose engine for analysis. By modifying the rules for the interpreter one can encode, in an agile way, the semantics of different languages, including logic, functional, and concurrent ones. As in [119], we also use program specialization to perform the so-called removal of the interpreter, but in addition, in the approach presented here we repeatedly use specialization for propagating the information about the constraints that occur in the initial configurations and in the error configurations. Also the class of the properties to be verified, which in the approach presented here is restricted to reachability properties, can be extended to those specified by more expressive logics, such as the Computational Tree Logic used in [62] for the verification of infinite state reactive systems.

A widely used verification technique implemented by software model checkers (e.g. SLAM and BLAST) is the *Counter-Example Guided Abstraction Refinement* (CEGAR) [90] which, given a program P and a correctness property φ, uses an abstract model $\alpha(P)$ to check whether or not P satisfies φ. If $\alpha(P)$ satisfies φ then P satisfies φ, otherwise a counterexample, i.e., an execution which makes the program incorrect, is produced. The counterexample is then analysed: if it turns out to be a real execution of P (*genuine* counterexample) then the program is proved to be incorrect, otherwise it has been generated due to a too coarse abstraction (*spurious* counterexample) and $\alpha(P)$ needs to be refined. The CEGAR approach has also been implemented by using CLP. In particular, in [123], the authors have designed a CEGAR-based software model checker for C programs, called ARMC. In [85], another CLP-based software model checker for C programs, called TRACER, is presented. It integrates an abstraction refinement phase within a symbolic execution process.

Our approach can be regarded as complementary to the approaches based on

CEGAR. Indeed, we begin by making no abstraction at all, and if the specialization process is deemed to diverge, then we perform some generalization steps which play a role similar to that of abstraction. (Note, however, that program specialization preserves program equivalence.) There are various generalization operators that we can apply for that purpose and by varying those operators we can tune the specialization process in the hope of making it more effective for the proofs of the properties of interest. Moreover, since our specialization-based method preserves the semantics of the original specification, we can apply a *sequence* of specializations, thereby refining the analysis and possibly improving the level of precision.

In the field of static program analysis the idea of performing backward and forward semantic analyses has been proposed in [31]. These analyses have been combined, for instance, in [33], to devise a fixpoint-guided abstraction refinement algorithm which has been proved to be at least as powerful as the CEGAR algorithm where the refinement is performed by applying a backward analysis. An enhanced version of that algorithm, which improves the abstract state space exploration and makes use of disjunctive abstract domains, has been proposed in [124]. In the approach presented here the idea of iterating program analysis and traversing the computation graph both in the forward and backward manner can be fruitfully exploited, once the program analysis task has been reduced, as we do, to a program transformation task.

CHAPTER 6

Verifying Array Programs

As already pointed out, one of the most appealing features of our approach is that it provides a very rich program verification framework where one can compose together several transformation strategies. Indeed, in the previous chapters we have provided different transformation strategies which can be used within the verification framework to prove partial correctness properties of imperative programs acting on integer variables. In this chapter we extend the strategy presented in Chapter 5 to prove partial correctness properties of programs manipulating arrays. This extension applies to the Verification Conditions Transformation step and is twofold. First, in order to specify verification conditions for array programs, we introduce the class of *constraint logic programs over integer arrays*, denoted CLP(Array). In particular, CLP(Array) programs may contain occurrences of `read` and `write` predicates that are interpreted as the input and output relations of the usual read and write operations on arrays. Secondly, in order to verify these verification conditions, besides the usual unfolding and folding rules, we consider the *constraint replacement* rule, which allows us to replace constraints by equivalent ones in the theory of arrays [20, 70, 111]. The transformation strategy may introduce some auxiliary predicates by using a generalization strategy that extends to CLP(Array) the generalization strategies for CLP programs over integers or reals.

This chapter is organized as follows. In Section 6.1 we introduce the class of CLP(Array) programs. Then, in Section 6.2 we present the oveall verification method, and in Section 6.3 we present the automatic strategy designed for applying the transformation rules with the objective of obtaining a proof (or a disproof) of the properties of interest. Finally, in Section 6.4, we present the result of the experimental evaluation obtained by applying the strategy on a set of array programs taken from the literature.

6.1 Constraint Logic Programs on Arrays

In this section we introduce the set CLP(Array) of CLP programs with constraints in the domain of integer arrays.

If $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and coefficients, then $\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, and $\mathtt{p}_1 > \mathtt{p}_2$ are *atomic integer constraints*. The dimension $\mathtt{n}$ of an array $\mathtt{a}$ is represented as a binary relation by the predicate $\mathtt{dim}(\mathtt{a}, \mathtt{n})$. For reasons of simplicity we consider one-dimensional arrays only. The read and write operations on arrays are represented by the predicates `read` and `write`, respectively, as follows: $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$ denotes the $\mathtt{i}$-th element of array $\mathtt{a}$ is the value $\mathtt{v}$, and $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$ denotes that the array $\mathtt{b}$ that is equal to the array $\mathtt{a}$ except that its $\mathtt{i}$-th element is $\mathtt{v}$. We assume that both indexes and values are integers, but our method is parametric with respect to the index and value domains. (Note, however, that the result of a verification task may depend on the constraint solver used, and hence on the constraint domain.)

An *atomic array constraint* is an atom of the following form: either $\mathtt{dim}(\mathtt{a}, \mathtt{n})$, or $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$, or $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$. A *constraint* is either `true`, or an atomic (integer or array) constraint, or a *conjunction* of constraints. An *atom* is an atomic formula of the form $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{m})$, where $\mathtt{p}$ is a predicate symbol not in $\{=, \geq, >, \mathtt{dim}, \mathtt{read}, \mathtt{write}\}$ and $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m}$ are terms constructed out of variables, constants, and function symbols different from `+` and `*`.

Now we define the semantics of CLP(Array) programs. An *$\mathcal{A}$-interpretation* is an interpretation I, that is, a set D, a function in $D^n \to D$ for each function symbol of arity n, and a relation on D^n for each predicate symbol of arity n, such that:

(i) the set D is the Herbrand universe [105] constructed out of the set $\mathbb{Z}$ of the integers, the constants, and the function symbols different from `+` and `*`,

(ii) I assigns to $+, *, =, \geq, >$ the usual meaning in $\mathbb{Z}$,

(iii) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$, for all integers $\mathtt{d}$,
$\mathtt{dim}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{d})$ is true in I iff $\mathtt{d} = \mathtt{n}$

(iv) I interprets the predicates `read` and `write` as follows: for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ and $\mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1}$ of integers, for all integers $\mathtt{i}$ and $\mathtt{v}$,
$\mathtt{read}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v})$ is true in I iff $0 \leq \mathtt{i} \leq \mathtt{n}-1$ and $\mathtt{v} = \mathtt{a}_\mathtt{i}$, and
$\mathtt{write}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v}, \mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1})$ is true in I iff
$0 \leq \mathtt{i} \leq \mathtt{n}-1$, $\mathtt{n} = \mathtt{m}$, $\mathtt{b}_\mathtt{i} = \mathtt{v}$, and for $\mathtt{j} = 0, \ldots, \mathtt{n}-1$, if $\mathtt{j} \neq \mathtt{i}$ then $\mathtt{a}_\mathtt{j} = \mathtt{b}_\mathtt{j}$

(v) I is an Herbrand interpretation [105] for function and predicate symbols different from `+`, `*`, `=`, $\geq$, `>`, `dim`, `read`, and `write`.

We can identify an $\mathcal{A}$-interpretation I with the set of ground atoms that are true in I, and hence $\mathcal{A}$-interpretations are partially ordered by set inclusion.

We write $\mathcal{A} \models \varphi$ if φ is true in every $\mathcal{A}$-interpretation. A constraint `c` is *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where in general, for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. Likewise, $\forall(\varphi)$ denotes the universal closure of φ. A constraint is *unsatisfiable* if it is not satisfiable. A constraint `c` *entails* a constraint `d`, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. By $vars(\varphi)$ we denote the free variables of φ.

We assume that we are given a solver to check the satisfiability and the entailment of constrains in $\mathcal{A}$. To this aim we can use any solver that implements algorithms for satisfiability and entailment in the theory of integer arrays [20, 70].

The semantics of a CLP(Array) program P is defined to be the *least* $\mathcal{A}$*-model* of P, denoted $M(P)$, that is, the least $\mathcal{A}$-interpretation I such that every clause of P is true in I.

Given a CLP(Array) program P and a ground goal `G` of the form `:-A`, $P \cup \{\mathtt{G}\}$ is satisfiable (or, equivalently, $P \not\models \mathtt{A}$) if and only if $\mathtt{A} \notin M(P)$. This property is a straightforward extension to CLP(Array) programs of van Emden and Kowalski's result [137].

6.2 The Verification Method

In this section we extend the Verification Conditions Generation (Step 2) and Verification Conditions Transformation (Step 3) steps of the verification method shown in Figure 8 to deal with array programs.

Let us consider an incorrectness triple of the form $\{\!\!\{\varphi_{init}\}\!\!\}$ `Prog` $\{\!\!\{\varphi_{error}\}\!\!\}$. We assume that the properties φ_{init} and φ_{error} can be expressed as conjunctions of (integer and array) constraints and `Prog` is a program acting on array variables. The imperative program `Prog` is correct with respect to the properties φ_{init} and φ_{error} iff $\mathtt{incorrect} \notin M(\mathtt{P})$ (or, equivalently, $\mathtt{P} \not\models \mathtt{incorrect}$), where $M(\mathtt{P})$ is the least $\mathcal{A}$-model of CLP program `P` encoding the given incorrectness triple (see Section 3.4).

In order to deal with array programs we extend the CLP interpreter `I` presented in Sections 3.2 and 3.3 by adding some extra clauses to the definition of the predicate `tr`. For instance, the following clause encodes the transition relation for the array assignment $\ell\!:\!a[ie]\!=\!e$

```
tr(cf(cmd(L,asgn(arrayelem(A,Ie),Ae),Lp),E), cf(cmd(Lp,C),Ep)) :-
   aeval(Ie,E,I), aeval(Ae,E,V), lookup(E,array(A),FA),
   write(FA,I,V,FAp), update(E,array(A),FAp,Ep), at(Lp,C).
```

In Step 2 the CLP Interpreter `I` is specialized with respect to the CLP Encoding `T` of the given incorrectness triple, thereby deriving a new program `V` representing the verification conditions for `Prog`.

The specialization of `I` is obtained by applying a variant of the $\mathit{Specialize}_{vcg}$ strategy presented in Chapter 3. The main difference is that the CLP programs considered in this chapter contain `read`, `write`, and `dim` predicates. The `read` and `write` predicates are never unfolded during specialization and they occur in the residual CLP(Array) program `V`. All occurrences of the `dim` predicate are eliminated by replacing them by suitable integer constraints on indexes.

Step 3 has the objective of checking, through further transformations, the satisfiability of the verification conditions generated by Step 2. In particular, the strategy aims at deriving either (i) a CLP(Array) program that has no constrained facts (hence proving satisfiability of the verification conditions and partial correctness of the program), or (ii) a CLP(Array) program containing the fact `incorrect` (hence proving that the verification conditions are unsatisfiable and the program does not satisfy the given property).

In this step we start with a program reversal of the CLP(Array) program `V`, then we proceed with a further program transformation based on the application of transformations that, under suitable conditions, preserve the least $\mathcal{A}$-model semantics of CLP(Array) programs. In particular, we perform the following transformations: `R` $:=$ *Reverse*(`V`); `S` $:=$ $\mathit{Transform}_{prop}$(`R`), during which we apply the following transformation rules: (i) definition introduction, (ii) unfolding, (iii) clause removal, (iv) *constraint replacement*, and (v) folding. These rules are an adaptation to CLP(Array) programs of the unfold/fold rules presented in Chapter 2 for a generic CLP language, and, by Theorem 1 we have that, by a sequence of applications of the above rules, `incorrect` $\in M($`V`$)$ iff `incorrect` $\in M($`S`$)$.

In particular, in order to deal with constraints in $\mathcal{A}$, the novel transformation strategy $\mathit{Transform}_{prop}$ makes use of the constraint replacement rule. The equivalences needed for constraint replacements are shown to hold in $\mathcal{A}$ by using a relational version of the theory of arrays with dimension [20, 70]. In particular, the constraint replacements we apply during the transformations described in Section 6.3 follow from the following axioms where all variables are universally quantified at the front:

(A1) `I`$=$`J`, `read(A,I,U)`, `read(A,J,V)` $\rightarrow$ `U`$=$`V`

(A2) $\mathtt{I}=\mathtt{J}$, write(A, I, U, B), read(B, J, V) $\rightarrow$ $\mathtt{U}=\mathtt{V}$

(A3) $\mathtt{I}\neq\mathtt{J}$, write(A, I, U, B), read(B, J, V) $\rightarrow$ read(A, J, V)

Axiom (A1) is often called *array congruence* and axioms (A2) and (A3) are collectively called *read-over-write*. We omit the usual axioms for dim.

We will describe this step in detail in Section 6.3. Before presenting the transformation strategy we give the reader a general idea of the method by verifying the correctness of the program for computing the maximum of an array. We apply the $\mathit{Transform}_{\mathit{prop}}$ strategy and we show how the constraint replacement rule can be used after the unfolding to exploit the theory of array encoded by mean of suitable replacement laws.

Example 8 (Computing the maximum of an array). Let us consider the following incorrectness triple $\{\!\{\varphi_{init}(i, n, a, max)\}\!\}$ arraymax $\{\!\{\varphi_{error}(n, a, max)\}\!\}$, where: (i) $\varphi_{init}(i, n, a, max)$ is $i \geq 0 \wedge n = dim(a) \wedge n \geq i+1 \wedge max = a[i]$, (ii) $\varphi_{error}(n, a, max)$ is $\exists k\ (0 \leq k < n \wedge a[k] > max)$, and arraymax is the program:

```
i=0;
while (x<n) {
  if (a[i] > max)
    max=a[i];
  i=i+1;
}
```

Listing 6.1: Program arraymax

If this triple holds, then the value of *max* computed by the program arraymax is not the maximal element of the given array a with n (≥ 1) elements.

We start off by executing the CLP Translation step which generates the CLP Encoding T of the given incorrectness triple. This construction is done as indicated in Section 3.1. and, in particular, the program arraymax is translated into a set of CLP facts defining the predicate at, and the formulas φ_{init} and φ_{error} are translated into the clauses for the predicates phiInit and phiError. We only list the fact on which the predicate initConf depends on, and the CLP clauses encoding φ_{init} and φ_{error}, respectively.

1. at(0,asgn(int(i),int(0),1)).[1]

[1] Note that the term 0 represents the label associated with i=0 in Listing 6.1. We recall that in the definition of the predicate at we encode the information about the control flow of the imperative program, for instance, the term 1 is the label where to jump after the execution of the command i=0

respectively, (ii) the environment E is the pair of lists ([[int(i),I], [int(n),N],[array(a

and (iii) the formulas φ_{init} and φ_{error}, are given by the following CLP clauses:

2. phiInit(E) :- I$\geq$0,N$\geq$I+1, read(A,I,Max).
3. phiError(E) :- K$\geq$0,N>K,Z>Max, read(A,K,Z).

Next, we apply the Verification Conditions Generation step of our verification method and we get the CLP program V representing the verification conditions for the arraymax program:

```
 4. incorrect :- I=0, N≥1, read(A,I,Max), new1(I,N,A,Max).
 5. new1(I,N,A,Max) :- I1=I+1, I<N, I≥0,M>Max, read(A,I,M),
      new1(I1,N,A,M).
 6. new1(I,N,A,Max) :- I1=I+1, I<N, I≥0, M≤Max, read((A,I,M),
      new1(I1,N,A,Max).
 7. new1(I,N,A,Max) :- I≥N,K≥0,N>K,Z>Max, read(A,K,Z).
```

We have that new1(I, N, A, Max) encodes the reachability of the error configuration from any initial configuration, where the program variables i, n, a, max are represented by I, N, A, Max, respectively.

In order to propagate the error property, similarly to the Example 6 of Section 5.2.1, we first 'reverse' program V and we get the following program $\mathtt{V}_{rev}$:

```
 8. incorrect :- b(U), r2(U).
 9. r2(V) :- trans(U,V), r2(U).
10. r2(U) :- a(U).
```

where predicates a(U), b(U), and trans(U,V) are defined as follows:

```
11. a([new1,I,N,A,Max]) :- I=0, N≥1, read(A,I,Max).
12. trans([new1,I,N,A,Max],[new1,I1,N,A,M]) :-
      I1=I+1, I<N, I≥0, M>Max, read(A,I,M).
13. trans([new1,I,N,A,Max],[new1,I1,N,A,Max]) :-
      I1=I+1, I<N, I≥0, M≤Max, read(A,I,M).
14. b([new1,I,N,A,Max]) :- I≥N, K≥0, K<N, Z>Max, read(A,K,Z).
```

Let us now apply the Verification Conditions Transformation step of the verification method.

In order to deal with the theory of arrays we make use of the constraint replacement rule together with the following law, which is a consequence of the fact that an array is a finite function:

```
(Law L1)  read(A,K,Z), read(A,I,M) ↔
                    ( K=I, Z=M, read(A,K,Z) )   ∨
```

(K≠I, read(A,K,Z), read(A,I,M))

In general, when applying the transformation strategy in the case of array programs, some additional laws may be required (see, for instance, [20]). For the Definition Introduction procedure we use a particular generalization operator, called *WidenSum* [62], which is a variant of the classical widening operator presented in Section 4.2.1 and behaves as follows. Given any atomic constraint a, let us denote by *sumcoeff*(a) the sum of the absolute values of the coefficients of a. Given any two constraints c and d, *WidenSum*(c, d) returns the conjunction of: (i) all atomic constraints a in c such that $d \sqsubseteq a$, and (ii) all atomic constraints b in d such that *sumcoeff*(b) $\leq$ *sumcoeff*(e) for some atomic constraint e in c.

Unfolding. We start off by unfolding clause 8 with respect to the atom `b(U)`, and we get:

```
15. incorrect :- I≥N, K≥0, K<N, Z>Max, read(A,K,Z),
      r2([new1,I,N,A,Max]).
```

The Clause Removal phase leaves unchanged the set of clauses we have derived so far. Since no clause in *Defs* can be used to fold clause 15 the Definition Introduction introduces the following clause:

```
16. new2(I,N,A,Max,K,Z) :- I≥N, K≥0, K<N, Z>Max, read(A,K,Z),
      r2([new1,I,N,A,Max]).
```

Now we proceed by performing a second iteration of the body of the while-loop of the transformation strategy because clause 16 is in *InCls*.

Unfolding. By unfolding clause 16 with respect to `r2([new1,I,N,A,Max])`, we get the following clauses:

```
17. new2(I,N,A,Max,K,Z) :- I≥N, K≥0, K<N, Z>Max, read(A,K,Z),
      trans(U,[new1,I,N,A,Max]), r2(U).
18. new2(I,N,A,Max,K,Z) :- I≥N, K≥0, K<N, Z>Max, read(A,K,Z),
      a([new1,I,N,A,Max]).
```

By unfolding clause 17 with respect to `trans(U,[new1,I,N,A,Max])`, we get:

```
19. new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,
      read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).
20. new2(I1,N,A,Max,K,Z) :- I1=I+1,N=I1,K≥0,K<I1,M≤Max,Z>Max,
      read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).
```

By unfolding clause 18 with respect to `a([new1,I,N,A,Max])`, we get an empty set of clauses (indeed, the constraint `I≥N, I=0, N≥1` is unsatisfiable).

Now we apply a subsidiary procedure, called Constraint Replacement, to deal with the theory of arrays. This phase performs the following two steps:
(i) it replaces the conjunction of atoms '`read(A,K,Z), read(A,I,M)`' occurring in the body of clause 19 by the right hand side of Law L1, and then
(ii) it splits the derived clause into the following two clauses, each of which corresponds to a disjunct of that right hand side.

21. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
    `K=I, Z=M, read(A,K,Z), r2([new1,I,N,A,Max]).`
22. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
    `K≠I, read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).`

Clause Removal. The constraint '`Z>M, Z=M`' in the body of clause 21 is unsatisfiable. Therefore, this clause is removed from `S`. From clause 22, by replacing '`K≠I`' by '`K<I v K>I`' and simplifying the constraints, we get:

23. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
    `read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).`

By performing from clause 20 a sequence of goal replacement and clause removal transformations similar to that we have performed from clause 19, we get the following clause:

24. `new2(I1,N,A,Max,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
    `read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).`

Definition Introduction. The comparison between the definition clause 16 we have introduced above, and clauses 23 and 24 which we should fold, shows the risk of introducing an unlimited number of definitions whose body contains the atoms `read((A,N),K,Z)` and `r2([new1,I,N,A,Max])`. Thus, in order to fold clauses 23 and 24, we introduce the following new definition:

25. `new3(I,N,A,Max,K,Z) :- K≥0, K<N, K<I, Z>Max, read(A,K,Z),`
    `r2([new1,I,N,A,Max]).`

The constraint in the body of this clause is obtained by generalizing: (i) the projection of the constraint in the body of clause 23 on the variables `I, N, A, Max, K, Z` (which are the variables of clause 23 that occur in the atoms `read(A,K,Z)` and `r2([new1,I,N,A,Max])`, and (ii) the constraint occurring in the body of clause 16. This generalization step can be seen as an application of the above mentioned *WidenSum* generalization operator.
Now we perform the third iteration of the body of the while-loop of the strategy.

Unfolding, Constraint Replacement, and Clause Removal. By unfolding, goal replacement, and clause removal, from clause 25 we get:

26. `new3(I1,N,A,M,K,Z) :- I1=I+1, K≥0, K<I, N≥I1, M>Max, Z>M,`
    `read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).`
27. `new3(I1,N,A,Max,K,Z) :- I1=I+1, K≥0, K<I, N≥I1, M≤Max, Z>Max,`
    `read(A,K,Z), read(A,I,M), r2([new1,I,N,A,Max]).`

Definition Introduction. In order to fold clauses 26 and 27, we do not need to introduce any new definition. Indeed, it is possible to fold these clauses by using clause 25.

Since no clause to be processed is left (because *InCls* $=\emptyset$), the transformation strategy exits the outermost while-loop (α), and starts the Folding phase. Finally, we get the program `S` which consists of the following set of clauses:

28. `incorrect :- I≥N, K≥0, K<N, Z>Max, new2(I,N,A,Max,K,Z).`
29. `new2(I1,N,A,Max,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
    `read(A,I,M), new3(I,N,A,Max,K,Z).`
30. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
    `read(A,I,M), new3(I,N,A,Max,K,Z).`
31. `new3(I1,N,A,M,K,Z) :- I1=I+1, K≥0, K<I, N≥I1, M>Max, Z>M,`
    `read(A,I,M), new3(I,N,A,Max,K,Z).`
32. `new3(I1,N,A,Max,K,Z) :- I1=I+1, K≥0, K<I, N≥I1, M≤Max, Z>Max,`
    `read(A,I,M), new3(I,N,A,Max,K,Z).`

obtained by folding clause 15 using clause 16, and by folding clauses 23, 24, 26 and 27 by using clause 25.

No clause in this set is a constrained fact, and hence by removal of useless clauses we get the final program `S` consisting of the empty set of clauses. Thus, *arraymax* is correct with respect to the properties φ_{init} and φ_{error}. □

6.3 The Transformation Strategy

As mentioned above, the verification conditions expressed as the CLP(Array) program `V` generated by Step 2 are satisfiable iff `incorrect` $\notin M($`S`$)$. Our verification method is based on the fact that by transforming the CLP(Array) program `V` using rules that preserve the least $\mathcal{A}$-model, we get a new CLP(Array) program `S` that expresses equisatisfiable verification conditions.

The Verification Conditions Transformation (Step 3) step has the objective of showing, through further transformations, that *either* the verification conditions `V` are satisfiable (that is, `incorrect` $\notin M($`V`$)$ and hence `Prog` is correct with respect to φ_{init} and φ_{error}), *or* they are unsatisfiable (that is, `incorrect` $\in M($`V`$)$

and hence `Prog` is not correct with respect to φ_{init} and φ_{error}). To this aim, Step 3 propagates the initial and/or the error properties so as to derive from program `V` a program `S` where the predicate `incorrect` is defined by either (A) the fact '`incorrect`' (in which case the verification conditions are unsatisfiable and `Prog` is incorrect), or (B) the empty set of clauses (in which case the verification conditions are satisfiable and `Prog` is correct). In the case where neither (A) nor (B) holds, that is, in program `S` the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact '`incorrect`', we cannot conclude anything about the correctness of `Prog`. However, similarly to what has been proposed in Chapter 5, in this case we can iterate Step 3, alternating the propagation of the initial and error properties, in the hope of deriving a program where either (A) or (B) holds. Obviously, due to undecidability limitations, it may be the case that we never get a program where either (A) or (B) holds.

Step 3 is performed by applying the unfold/fold transformation rules according to the *Transform*$_{prop}$ strategy shown in Figure 12. *Transform*$_{prop}$ can be viewed as a backward propagation of the error property. The forward propagation of the initial property can be obtained by combining *Transform*$_{prop}$ with the *Reversal* transformation described in Section 5.2.3

Unfolding performs one inference step backward from the error property.

The Constraint Replacement phase, by applying the theory of arrays, infers new constraints on the variables of the only atom that occurs in the body of each clause obtained by the Unfolding phase. It works as follows. We select a clause, say `H :- c, G`, in the set *TransfC* of the clauses obtained by unfolding, and we replace that clause by the one(s) obtained by applying as long as possible the following rules. Note that this process always terminates and, in general, it is nondeterministic.

(RR1) If $\mathtt{c} \sqsubseteq (\mathtt{I}=\mathtt{J})$ then
replace: `read(A,I,U), read(A,J,V)` by: `U=V, read(A,I,U)`

(RR2) If $\mathtt{c} \equiv (\mathtt{read(A,I,U)}, \mathtt{read(A,J,V)}, \mathtt{d})$, $\mathtt{d} \not\sqsubseteq (\mathtt{I}\neq\mathtt{J})$, and $\mathtt{d} \sqsubseteq (\mathtt{U}\neq\mathtt{V})$
add to `c` then the constraint: $\mathtt{I}\neq\mathtt{J}$

(WR1) If $\mathtt{c} \sqsubseteq (\mathtt{I}=\mathtt{J})$ then
replace: `write(A,I,U,B), read(B,J,V)`
by: `U=V, write(A,I,U,B)`

(WR2) If $\mathtt{c} \sqsubseteq (\mathtt{I}\neq\mathtt{J})$
replace: `write(A,I,U,B), read(B,J,V)`
by: `write(A,I,U,B), read(A,J,V)`

Input: Program R.
Output: Program S such that incorrect $\in M$(R) iff incorrect $\in M$(S).

```
INITIALIZATION:
S := ∅;
$InCls$ := { incorrect:- c$_1$(X), A$_1$(X), ..., incorrect:- c$_j$(X), A$_j$(X) };
$Defs$ := ∅;
  ($\alpha$) while in $InCls$ there is a clause $C$ that is not a constrained fact do
    UNFOLDING:
      $TransfC := Unf(C, A)$, where $A$ is the leftmost atom in the body of $C$;
    CONSTRAINT REPLACEMENT:
      ($\alpha 2$) while in $TransfC$ there is a clause $E$ such that it is possible
                   to apply a constraint replacement rule $R$ in $Rules$ do
                 $TransfC := (TransfC - \{E\}) \cup R(E)$
               end-while;
    CLAUSE REMOVAL:
      ($\alpha 3$) while in $TransfC$ there are two distinct clauses $E_1$ and $E_2$
                   such that $E_1$ subsumes $E_2$ do
                 $TransfC := TransfC - \{E_2\}$;
               end-while;
    DEFINITION INTRODUCTION:
      ($\alpha 4$) while in $TransfC$ there is a clause $E$ that is not a constrained fact
                   and cannot be folded using a definition in $Defs$ do
                 $G := Gen(E, Defs)$;
                 $Defs := Defs \cup \{child(G, C)\}$;
                 $InCls := InCls \cup \{G\}$;
               end-while;
    $InCls := InCls - \{C\}$;   S := S $\cup$ $TransfC$;
  end-while;
FOLDING:
  ($\beta$) while in S there is a clause $E$ that can be folded
           by a clause $D$ in $Defs$ do
         S := (S $- \{E\}) \cup \{F\}$, where $F$ is derived by folding $E$ using $D$;
       end-while;
Remove from $P_t$ all clauses for predicates on which incorrect does not depend.
```

Figure 12: The *Transform*$_{prop}$ strategy

(WR3) If $\mathtt{c} \not\sqsubseteq \mathtt{I}=\mathtt{J}$ and $\mathtt{c} \not\sqsubseteq \mathtt{I}\neq\mathtt{J}$ then

```
replace: H :- c, write(A,I,U,B), read(B,J,V), G
by:      H :- c, I=J, U=V, write(A,I,U,B), G
   and   H :- c, I≠J, write(A,I,U,B), read(A,J,V), G
```

Rules RR1 and RR2 are derived from the array axiom A1 (see Section 6.2), and rules WR1–WR3 are derived from the array axioms A2 and A3 (see Section 6.2). Let *Rules* be a set of the constraint replacement rules. We denote by $R(E)$ the set of clauses obtained by applying a rule R in *Rules* to the constraint occurring in the body of clause E.

The DEFINITION INTRODUCTION phase introduces new predicate definitions by suitable generalizations of the constraints. These generalizations guarantee the termination of *Transform*$_{prop}$, but at the same time they should be as specific as possible in order to achieve maximal precision. This phase works as follows. Let $C1$ in *TransfC* be a clause of the form `H :- c, p(X)`. If in *Defs* there is an ancestor D: `newp(X) :- d, p(X)` of clause $C1$ such that *vars*(`d`) $\subseteq$ *vars*(`c`) and `c` $\sqsubseteq$ `d`, then we fold $C1$ using D. Otherwise, we introduce a clause of the form `newp(X) :- gen, p(X)` where: (i) `newp` is a predicate symbol occurring neither in the initial program nor in *Defs*, and (ii) `gen` is a constraint such that *vars*(`gen`) $\subseteq$ *vars*(`c`) and `c` $\sqsubseteq$ `gen`. The constraint `gen` is called a generalization of the constraint `c` and is constructed as follows.

Let `c` be of the form $\mathtt{i}_1, \mathtt{rw}_1$, where $\mathtt{i}_1$ is an integer constraint and $\mathtt{rw}_1$ is a conjunction of `read` and `write` constraints. Without loss of generality, we assume that all occurrences of integers in `read` constraints of `c` are distinct variables not occurring in `X` (this condition can always be fulfilled by adding suitable integer equalities).

(1) Delete all `write` constraints from $\mathtt{rw}_1$, hence deriving $\mathtt{r}_1$.

(2) Compute the projection $\mathtt{i}_2$ (in the rationals $\mathbb{Q}$) of the constraint $\mathtt{i}_1$ onto *vars*$(\mathtt{r}_1) \cup \{\mathtt{X}\}$. (Recall that the projection in $\mathbb{Q}$ of a constraint $\mathtt{c(Y,Z)}$ onto the tuple `Y` of variables is a constraint $\mathtt{c_p(Y)}$ such that $\mathbb{Q} \models \forall\, \mathtt{Y}(\mathtt{c_p(Y)} \leftrightarrow \exists\, \mathtt{Z}\, \mathtt{c(Y,Z)})$.)

(3) Delete from $\mathtt{r}_1$ all `read(A,I,V)` constraints such that either (i) `A` does not occur in `X`, or (ii) `V` does not occur in $\mathtt{i}_2$, thereby deriving a new value for $\mathtt{r}_1$. If at least one `read` has been deleted during this step, then go to Step 2.

(4) Let $\mathtt{i}_2, \mathtt{r}_2$ be the constraint obtained after the possibly repeated executions of Steps 2–3.
If in *Defs* there is an ancestor (defined as the reflexive, transitive closure of the parent relation) of C of the form $\mathtt{H_0} \text{ :- } \mathtt{i_0, r_0, p(X)}$ such that $\mathtt{r_0, p(X)}$ is a subconjunction of $\mathtt{r_2, p(X)}$,

then compute a generalization $\mathtt{g}$ of the constraints $\mathtt{i}_2$ and $\mathtt{i}_0$ such that $\mathtt{i}_2 \sqsubseteq \mathtt{g}$, by using a *generalization operator* for linear constraints (see Section 4.2.1). Define the constraint $\mathtt{gen}$ as $\mathtt{g}, \mathtt{r}_0$;
else define the constraint $\mathtt{gen}$ as $\mathtt{i}_2, \mathtt{r}_2$.

6.3.1 Termination and Soundness of the Transformation Strategy

The following theorem establishes the termination and soundness of the *Transform*$_{prop}$ strategy.

Theorem 8 (Termination and Soundness of the *Transform*$_{prop}$ strategy). (i) The *Transform*$_{prop}$ strategy terminates. (ii) Let program `S` be the output of the *Transform*$_{prop}$ strategy applied on the input program `R`. Then, $\mathtt{incorrect} \in M(\mathtt{R})$ iff $\mathtt{incorrect} \in M(\mathtt{S})$.

Proof. (i) The termination of the *Transform*$_{prop}$ strategy is based on the following facts:
(1) Constraint satisfiability and entailment are checked by a terminating solver (note that completeness is not necessary for the termination of *Transform*$_{prop}$).
(2) Constraint Replacement terminates (see above).
(3) The set of new clauses that, during the execution of the *Transform*$_{prop}$ strategy, can be introduced by Definition Introduction steps is finite. Indeed, by construction, they are all of the form $\mathtt{H} \;\mathtt{:-}\; \mathtt{i}, \mathtt{r}, \mathtt{p(X)}$, where: (3.1) `X` is a tuple of variables, (3.2) `i` is an integer constraint, (3.3) `r` is a conjunction of array constraints of the form $\mathtt{read(A,I,V)}$, where `A` is a variable in `X` and the variables `I` and `V` occur in `i` only, (3.4) the cardinality of `r` is bounded, because generalization does not introduce a clause $\mathtt{newp(X)} \;\mathtt{:-}\; \mathtt{i}_2, \mathtt{r}_2, \mathtt{p(X)}$ if there exists an ancestor clause of the form $\mathtt{H}_0 \;\mathtt{:-}\; \mathtt{i}_0, \mathtt{r}_0, \mathtt{p(X)}$ such that $\mathtt{r}_0, \mathtt{p(X)}$ is a subconjunction of $\mathtt{r}_2, \mathtt{p(X)}$, (3.5) we assume that the generalization operator on integer constraints has the following *finiteness* property: only finite chains of generalizations of any given integer constraint can be generated by applying the operator. The already mentioned generalization operators presented in Section 4.2.1 satisfy this finiteness property.

(ii) The correctness of the strategy with respect to the least $\mathcal{A}$-model semantics follows from Theorem 1, by observing that every clause defining a new predicate introduced by Definition Introduction is unfolded once during the execution of the strategy (indeed every such clause is added to *InCls*). □

Example 9 (Array sequence initialization). Let us consider the following incorrectness triple $\{\!\{\varphi_{init}(i,n,a)\}\!\}$ *SeqInit* $\{\!\{\varphi_{error}(n,a)\}\!\}$ where: (i) $\varphi_{init}(i,n,a)$

is $i \geq 0 \wedge n = dim(a) \wedge n \geq 1$, (ii) $\varphi_{error}(n, a)$ is $\exists j\ (0 \leq j \wedge j+1 < n \wedge a[j] \geq a[j+1])$, and (iii) `seqinit` is the imperative program listed below:

```
i=1;
while (i<n) {
  a[i]=a[i-1]+1;
  i=i+1;
}
```

Listing 6.2: Program `seqinit`

which initializes a given array a of n integers by the sequence: $a[0]$, $a[0]+1$, ..., $a[0]+n-1$.

First, by applying the CLP Translation step, the above incorrectness triple is translated into a CLP(Array) program `T`. In particular, the predicates `phiInit` and `phiError` are defined by the following clauses:

5. `phiInit(E) :- I≥0, dim(A,N), N≥1.`
6. `phiError(E) :- Z=W+1, W≥0, W+1<N, U≥V, read(A,W,U), read(A,Z,V).`

representing the properties φ_{init} and φ_{error} respectively.

Now by applying the Verification Conditions Generation, we generate the following CLP program:

```
7. incorrect :-  Z=W+1, W≥0, W+1<N, U≥V, N≤I,
     read(A,W,U), read(A,Z,V), new1(I,N,A).
8. new1(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1,
     read(A,D,U), write(A,I,V,B), new1(I,N,A).
9. new1(I,N,A) :- I=1, N≥1.
```

The CLP(Array) program `R` expresses the verification conditions for `seqinit`. For reasons of simplicity, the predicates expressing the assertions associated with assignments and conditionals have been unfolded away during the removal of the interpreter.

Due to the presence of integer and array variables, the least $\mathcal{A}$-model $M(\mathtt{R})$ may be infinite, and both the bottom-up and top-down evaluation of the goal `:- incorrect` may not terminate (indeed, this is the case in our example above). Thus, we cannot directly use the standard CLP systems to prove program correctness. In order to cope with this difficulty, we apply the *Transform*$_{prop}$ to `R` which allows us to avoid the exhaustive exploration of the possibly infinite space of reachable configurations.

Unfolding. First, we unfold clause 7 w.r.t. the atom `new1(I,N,A)`, and we get:

10. `incorrect :- Z=W+1, W≥0, Z≤I, D=I−1, N=I+1, Y=X+1, U≥V,`
    `read(B,W,U), read(B,Z,V), read(A,D,X), write(A,I,Y,B), new1(I,N,A).`

CONSTRAINT REPLACEMENT. Then, by applying the replacement rules WR2, WR3, and RR1 to clause 10, we get the following clause:

11. `incorrect :- Z=W+1, W≥0, Z<I, D=I−1, N=I+1, Y=X+1, U≥V,`
    `read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), new1(I,N,A).`

In particular, since `W≠I` is entailed by the constraint in clause 10, we apply rule WR2 and we obtain a new clause, say 11.1, where `read(B,W,U), write(A,I,Y,B)` is replaced by `read(A,W,U), write(A,I,Y,B)`. Then, since neither `Z=I` nor `Z≠I` is entailed by the constraint in clause 11.1, we apply rule WR3 and we obtain two clauses 11.2 and 11.3, where the constraint `read(B,Z,V), write(A,I,Y,B)` is replaced by `Z = I, Y = V , write(A,I,Y,B)` and `Z ≠ I, read(A,Z,U), write(A,I,Y,B)`, respectively. Finally, since `D = W` is entailed by the constraint in clause 11.3, we apply rule RR1 to clause 11.3 and we add the constraint `X = U` to its body, hence deriving the unsatisfiable constraint `X = U, Y = X+1, Y = V, U≥V`. Thus, the clause derived by the latter replacement is removed. Clause 11 is derived from 11.3 by rewriting `Z≤I, Z ≠ I` as `Z<I`.

DEFINITION INTRODUCTION. In order to fold clause 11 we introduce a new definition by applying Steps (1)–(4) of the DEFINITION INTRODUCTION phase. In particular, by deleting the `write` constraint (1) and projecting the integer constraint (2), we get a constraint where the variable `X` occurs in `read(A,D,X)` only. Thus, we delete `read(A,D,X)` (3). Finally, we compute a generalization of the constraints occurring in clauses 7 and 11 by using the convex hull operator (4). We get:

12. `new2(I,N,A) :- Z=W+1, W≥0, N≤I+1, N≥W+2, W≤I−2, U≥V,`
    `read(A,W,U), read(A,Z,V), new1(I,N,A).`

Now we proceed by performing a second iteration of the body of the while-loop of the *Transform*$_{prop}$ strategy because *InCls* is not empty (indeed, at this point clause 12 belongs to *InCls*).

UNFOLDING. After unfolding clause 12 we get the following clause:

13. `new2(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,`
    `Z≤I, Z≥1, N>I, U≥V, read(B,W,U), read(B,Z,V),`
    `read(A,D,X), write(A,I,Y,B), new1(I,N,A).`

CONSTRAINT REPLACEMENT. Then, by applying rules RR1, WR2, and WR3 to clause 13, we get the following clause:

14. `new2(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,`

```
     Z<I, Z≥1, N>I, U≥V, read(A,W,U),  read(A,Z,V),
     read(A,D,X), write(A,I,Y,B), new1(I,N,A).
```

Definition Introduction. In order to fold clause 14 we introduce the following clause, whose body is derived by computing the widening [30, 34] of the integer constraints in the ancestor clause 12 with respect to the integer constraints in clause 14:

```
15. new3(I,N,A) :- Z=W+1, W≥0, W≤I−1, N>Z, U≥V,
     read(A,W,U), read(A,Z,V), new1(I,N,A).
```

Now we perform the third iteration of the body of the while-loop of the strategy starting from the newly introduced definition, that is, clause 15. After some unfolding and constraint replacement steps, from clause 15 we get:

```
16. new3(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, I≥1,
     Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),
     read(A,D,X), write(A,I,Y,B), new1(I,N,A).
```

The final transformed program is made out of the following clauses:

```
17. incorrect :- Z=W+1, W≥0, Z<I, D=I−1, N=I+1, Y=X+1, U≥V,
     read(B,W,U), read(B,Z,V), read(A,D,X), write(A,I,Y,B), new2(I,N,A).
18. new2(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,
     Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),
     read(A,D,X), write(A,I,Y,B), new3(I,N,A).
19. new3(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, I≥1,
     Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),
     read(A,D,X), write(A,I,Y,B), new3(I,N,A).
```

obtained by folding clause 11 using clause 12, and by folding clauses 14 and 16 by using clause 15.

Since this program has no constrained facts, by the last step of the $\mathit{Transform}_{prop}$ procedure we derive the empty program `S`, and we conclude that the program `seqinit` is correct with respect to the given φ_{init} and φ_{error} properties. □

6.4 Experimental Evaluation

We have performed an experimental evaluation of our method by using the VeriMAP tool (see Chapter 8) on a benchmark set of programs acting on arrays, mostly taken from the literature [17, 52, 77, 98]. The results of our experiments, which are summarized in Tables 6.1 and 6.2, show that our approach is effective and quite efficient in practice.

We now briefly discuss the programs we have used for our experimental evaluation (see Table 6.1 where we have also indicated the properties we have verified).

Program	*Code*	*Verified Property*

init	`for(i=0; i<n; i++)`
	`  a[i]=c;`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to a[i] = c$

init-partial	`for(i=0; i<k; i++)`
	`  a[i]=0;`
		$\forall i.\ (0 \leq i \wedge i < k \wedge k \leq n)$
		$\to a[i] = 0$

init-non-constant	`for(i=0; i<n; i++)`
	`  a[i]=2*i+c;`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to a[i] = 2 * i + c$

init-sequence	`a[0]=7;`
	`i=1;`
	`while(i<n) {`
	`  a[i]=a[i-1]+1;`
	`  i++;`
	`}`
		$\forall i.\ (1 \leq i \wedge i < n)$
		$\to a[i] = a[i-1] + 1$

copy	`for(i=0; i<n; i++)`
	`  a[i]=b[i];`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to a[i] = b[i]$

copy-partial	`for(i=0; i<k; i++)`
	`  a[i]=b[i];`
		$\forall i.\ (0 \leq i \wedge i < k \wedge k \leq n)$
		$\to a[i] = b[i]$

copy-reverse	`for(i=0; i<n; i++)`
	`  b[i]=a[i];`
	`for(i=0; i<n; i++)`
	`  a[i]=b[n-i-1];`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to a[i] = b[n-i-1]$

max	`m=a[0];`
	`i=1;`
	`while(i<n) {`
	`  if(a[i]>m)`
	`    m=a[i];`
	`  i++;`
	`}`
		$\forall i.\ (0 \leq i \wedge i < n \wedge n \geq 1)$
		$\to m \geq a[i]$

sum	`for(i=0; i<n; i++)`
	`  c[i]=a[i]+b[i];`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to c[i] = a[i] + b[i]$

difference	`for(i=0; i<n; i++)`
	`  c[i]=a[i]-b[i];`
		$\forall i.\ (0 \leq i \wedge i < n)$
		$\to c[i] = a[i] - b[i]$

find	`p=-1;`
	`for(i=0; i<n; i++)`
	` if(a[i]==e) {`
	`  p=i;`
	`  break;`
	`}`
		$(0 \leq p \wedge p < n)$
		$\rightarrow a[p] = e$

first-not-null	`s=n;`
	`for(i=0; i<n; ++i)`
	` if(s==n && a[i]!=0)`
	`  s=i;`
		$(0 \leq s \wedge s < n) \rightarrow$
		$(a[s] \neq 0 \wedge$
		$(\forall i.\ (0 \leq i \wedge i < s) \rightarrow$
		$a[i] = 0) \quad)$

find-first-non-null	`p=-1;`
	`for(i=0; i<n; i++)`
	` if(a[i]!=0) {`
	`  p=i;`
	`  break;`
	`}`
		$(0 \leq p \wedge p < n)$
		$\rightarrow a[p] \neq 0$

partition	`i=0;`
	`j=0;`
	`k=0;`
	`while(i<n) {`
	` if(a[i]>=0) {`
	`  b[j]=a[i];`
	`  j++;`
	` } else {`
	`  c[k]=a[i];`
	`  k++;`
	` }`
	` ++i;`
	`}`
		$(\forall i.\ (0 \leq i \wedge i < j)$
		$\rightarrow b[i] \geq 0) \quad \wedge$
		$(\forall i.\ (0 \leq i \wedge i < k)$
		$\rightarrow c[i] < 0)$

insertionsort-inner	`x=a[i];`
	`j=i-1;`
	`while(j>=0 && a[j]>x) {`
	` a[j+1]=a[j];`
	` --j;`
	`}`
		$\forall k.\ (0 \leq i \wedge i < n \wedge$
		$j{+}1 < k \wedge k \leq i) \rightarrow a[k] > x$

bubblesort-inner	`for(j=0; j<n-i-1; j++) {`
	` if(a[j] > a[j+1]) {`
	`  tmp = a[j];`
	`  a[j] = a[j+1];`
	`  a[j+1] = tmp;`
	` }`
	`}`
		$\forall k.\ (0 \leq i \wedge i < n \wedge 0 \leq k \wedge k < j \wedge j = n\text{–}i\text{–}1) \rightarrow a[k] \leq a[j]$

selectionsort-inner	`for(j=i+1; j<n; j++) {`
	` if(a[i]>a[j]) {`
	`  tmp=a[i];`
	`  a[i]=a[j];`
	`  a[j]=tmp;`
	` }`
	`}`
		$\forall k.(0 \leq i \wedge i \leq k \wedge k < n) \rightarrow a[k] \geq a[i]$

Table 6.1: Benchmark array programs. Variables `a,b,c` are arrays of integers of size `n`.

Some programs deal with array initialization: program *init* initializes all the elements of the array to a constant, while *init-non-constant* and *init-sequence* use expressions which depend on the element position and on the preceding element, respectively. Program *init-partial* initializes only an initial portion of the array. Program *copy* performs the element-wise copy of an entire array to another array, while *copy-partial* copies only an initial portion of the array, and the program *copy-reverse* copies the array in reverse order. The program *max* computes the maximum of an array. The programs *sum* and *difference* perform the element-wise sum and difference, respectively, of two input arrays. The program *find* looks for a particular value inside an array and returns the position of its first occurrence, if any, or a negative value otherwise. The programs *find-first-non-null* and *first-not-null* are two programs which return the position of the first non-zero element. For these programs, differently from [52, 77], we prove that when the search succeeds, the returned position contains a non-zero element and we also proved that all the preceding elements are zero elements. The program *partition* copies non-negative and negative elements of the array into two distinct arrays. The programs *insertionsort-inner*, *bubblesort-inner*, and *selectionsort-inner* are based on textbook implementations of sorting algorithms.

The source code of all the verification problems we have considered is available at `http://map.uniroma2.it/smc/vmcai/`.

For verifying the above programs we have applied the *Transform*$_{prop}$ strategy using different generalization operators, which are based on the widening and convex hull operators. In particular the Gen_W and Gen_S operators use the *Widen* and *CHWidenSum* operators between constraints [62].

We have also combined these operators with a delay mechanism which, before starting the actual generalization process, introduces a definition which is computed by using convex hull alone, without widening. We denote by Gen_{WD} and Gen_{SD} the operators obtained by combining delayed generalization with the *Widen* and *CHWidenSum* operators, respectively.

In Table 6.2 we report the results obtained by applying *Transform*$_{prop}$ with the four generalization operators mentioned above. The first column contains references to papers where the program verification example has been considered.

The last four columns are labeled with the name of the generalization operator. For each program proved correct we report the time in seconds taken to verify the property of interest. By *unknown* we indicate that *Transform*$_{prop}$ derives a CLP(Array) program containing constrained facts different from '`incorrect`', and hence the satisfiability (or the unsatisfiability) of the corresponding verification conditions cannot be checked.

We also report, for each generalization operator, the number of successfully verified programs (which measures the *precision* of the operator), the *total time* taken to run the whole benchmark and the *average time* per successful answer, respectively.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

The data presented in Table 6.2 show that by using the Gen_W operator, which applies the widening operator alone, our method is only able to prove 7 programs out of 17. However, precision can be recovered by applying the convex hull operator when introducing new definitions, possibly combined with widening.

The best trade-off between precision and performance is provided by the Gen_{WD} operator which is able to prove all 17 programs with an average time of $0.92\,s$. In this case the use of the delay mechanism, which uses convex hull, suffices to compensate the weakness demonstrated by the use of widening alone. Note also that one program, *init-sequence*, can only be proved by applying operators which use delayed generalization. This confirms the effectiveness of the convex hull operator which may help inferring relations among program variables,

Program	*References*	Gen_W	Gen_{WD}	Gen_S	Gen_{SD}
init	[17, 52, 130]	*unknown*	0.06	0.10	0.08
init-partial	[17, 52]	*unknown*	0.06	0.07	0.08
init-non-constant	[17, 52, 98, 130]	*unknown*	0.06	0.22	0.22
init-sequence	[77, 98]	*unknown*	0.80	*unknown*	1.20
copy	[17, 52, 77, 98, 130]	*unknown*	0.27	0.33	0.29
copy-partial	[17, 52]	*unknown*	0.29	0.34	0.34
copy-reverse	[17, 52]	*unknown*	0.27	0.46	0.45
max	[77, 98]	*unknown*	0.31	0.24	0.33
sum		*unknown*	0.68	1.14	1.12
difference	[17]	*unknown*	0.66	1.15	1.11
find	[17, 52]	0.25	0.43	0.46	0.45
first-not-null	[77]	0.38	0.41	0.42	0.42
find-first-non-null	[17, 52]	1.24	1.87	1.94	1.93
partition	[52, 98, 130]	0.06	0.11	0.14	0.12
insertionsort-inner	[77, 98, 130]	0.21	0.26	0.45	0.43
bubblesort-inner		2.46	2.71	2.45	2.75
selectionsort-inner	[130]	7.20	6.40	7.23	7.16

precision	7	17	16	17
total time	11.80	15.65	17.14	18.48
average time	1.69	0.92	1.07	1.09

Table 6.2: Verification results using the MAP system with different generalization operators. Times are in seconds.

and may ease the discovery of useful program invariants, while determining (in our set of examples) only a slight increase of verification times.

A detailed comparison of the performance of our system with respect to the other verification systems referred to in Table 6.1 is difficult to make at this time because the systems are not all readily available and also the results reported in the literature do not refer to the same code for the input C programs.

6.5 Related Work

The verification method presented in this chapter is also related to several other methods that use abstract interpretation and theorem proving techniques.

Now we briefly report on related papers which use abstract interpretations for finding invariants of programs that manipulate arrays. In [77], which builds upon [72], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is followed in [32] where a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs is introduced. In [64, 97] a predicate abstraction for inferring universally quantified properties of array elements is presented, and in [75] the authors present a similar technique which uses template-based quantified abstract domains.

Methods based on abstract interpretation construct overapproximations, that is, invariants implied by the program executions.

In [77], which builds upon [72], relational properties among array elements are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. This approach offers a compromise between the efficiency of *array smashing* (where the whole array is represented by a single abstract variable) and the precision of *array expansion* [18] (where every array element is associated with a distinct abstract variable). In [32] the authors present a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs based on slicing. In [75] a powerful technique using template-based quantified abstract domains, is applied to successfully generate quantified invariants. Other authors (see [64, 97]) use indexed predicate abstraction for inferring universally quantified properties about array elements. Methods based on abstract interpretation and predicate abstraction have the advance of being quite efficient because it fixes in advance a finite set of assertions where the invariants are searched for, but for the same reason it may lack flexibility as the abstraction should be re-designed when the verification fails.

Also theorem provers have been used for discovering invariants in programs which manipulate arrays and prove verification conditions generated from the programs. In particular, in [20] a satisfiability decision procedure for a decidable fragment of a theory of arrays is presented. That fragment is expressive enough to prove properties such as sortedness of arrays. In [92, 95, 112] the authors present some techniques based on theorem proving which may generate array invariants. In [130] a backward reachability analysis based on predicate abstraction and abstraction refinement is used for verifying assertions which are universally quantified over array indexes. Finally, we would like to mention that techniques based on Satisfiability Modulo Theory (SMT) have been applied for generating and verifying universally quantified properties over array variables (see, for instance, [3, 98]).

The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation because no finite set of abstractions is fixed in advance, but the suitable assertions needed by the proof are generated on the fly.

CHAPTER 7

Recursively Defined Properties

In Chapter 6 we have shown that our verification method can be extended toward a more general and powerful one which is able to deal with programs manipulating arrays.

In this chapter we make a step forward and we show that the verification method can also be used in the case when the initial and error properties are specified by sets of CLP clauses, rather than by constraints only. We also develop an example to exhibit that our method can be applied in a rather systematic way, and is amenable to automation by transferring to the field of program verification many techniques developed in the field of program transformation.

In particular, we extend the transformation strategy presented in Figure 12 with more general, semantics preserving, unfold/fold transformation rules for CLP programs. Indeed, during the Verification Conditions Transformation step, where we transform the verification conditions to propagate the constraints representing the initial and error properties, we make use of transformation rules that are more powerful than those used by program specialization. These rules include the *conjunctive definition* and the *conjunctive folding* rules and they allow us to introduce and transform new predicate definitions that correspond to *conjunctions* of old predicates, while program specialization can deal only with new predicates that correspond to specialized versions of *one* old predicate. These more powerful rules allow us to verify programs with respect to complex initial and error properties defined by sets of CLP clauses (for instance, recursively defined relations among program variables), whereas program specialization can only deal with initial and error properties specified by (sets of) constraints.

The transformation strategy used here is an extension of the transformation

```
UNFOLDING:
  $TransfC := Unf(C, A)$, where $A$ is the leftmost atom in the body of $C$;
  ($\alpha 1$) while in $TransfC$ there is a clause $D$ whose body contains an atom $A$
          that has not been derived by an atom with the same predicate of $A$ do
            $TransfC := (TransfC - \{D\}) \cup Unf(D, A)$;
     end-while;
```

Figure 13: The extended UNFOLDING phase of the *Transform*$_{prop}$

strategy *Transform*$_{prop}$ presented in Figure 12.

In order to deal with properties which are specified by sets of CLP clauses, we modify the UNFOLDING phase as shown in Figure 13. This extension ensures the termination of the *Unf* function whenever in the body of clauses occurs an atom whose predicate is defined by a set of recusively defined clauses.

Finally, during the DEFINITION INTRODUCTION phase of the *Transform*$_{prop}$ strategy, we allow ourselves to introduce new predicates by using definition clauses of the form: `newp :- c, G`, where `newp` is an atom with a new predicate symbol, `c` is a constraint, and `G` is a *conjunction of one or more atoms.* (Note that the new predicate definitions introduced during the verification example of the previous section are all of the form: `newp :- c, A`, where `A` is a single atom.) Clauses of that form will then be used for performing the FOLDING phase, which, similarly, is extended to deal with conjunction of atoms.

We show through the following example how the extended transformation strategy can be used to prove that the imperative `GCD` program, whose intended behavior is to compute the greatest common divisor of two positive integers, yields an integer which indeed satisfies the definition of greater common divisor, specified by using a set of (recursively defined) CLP clauses.

Example 10 (Greatest Common Divisor). Let us consider the following program incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ `GCD` $\{\!\{\varphi_{error}\}\!\}$ where: (i) $\varphi_{init}(m, n)$ is $m \geq 1 \wedge n \geq 1$, (ii) $\varphi_{error}(m, n, z)$ is $\exists\, d\,(gcd(m, n, d) \wedge d \neq z)$, and (iii) `GCD` is the following imperative program

```
x=m; y=n;
while (x!=y) {
  if (x>y)
    x=x-y;
  else
    y=y-x;
}
```

```
    z=x;
```

Listing 7.1: Program GCD

for computing the *greatest common divisor* z of two positive integers m and n.

The property φ_{error} uses the ternary predicate `gcd` defined by the following CLP clauses:

1. `gcd(X,Y,D) :- X>Y, X1=X-Y, gcd(X1,Y,D).`
2. `gcd(X,Y,D) :- X<Y, Y1=Y-X, gcd(X,Y1,D).`
3. `gcd(X,Y,D) :- X=Y, Y=D.`

Thus, the incorrectness triple holds if and only if, for some positive integers m and n, the program GCD computes a value of z that is different from the greatest common divisor of m and n.

As indicated in Section 3.1, the program GCD can be translated into a set of CLP facts defining the predicate `at`, but we will not show them here. The predicates `phiInit` and `phiError` are defined as follows:

4. `phiInit(E) :-M≥1, N≥1.`
5. `phiError(E) :- gcd(M,N,D), D≠Z.`

where `E` is the term
`([[int(m),M],[int(n),N],[int(x),X],[int(y),Y], [int(z),Z]],[])`
encoding the environment.

Now, by performing the Verification Conditions Generation step[1] of our verification method we derive the following CLP program:

6. `incorrect :- M≥1,N≥1, new1(M,N,M,N,Z).`
7. `new1(M,N,X,Y,Z) :- X>Y,X1=X-Y, new1(M,N,X1,Y,Z).`
8. `new1(M,N,X,Y,Z) :- X<Y,Y1=Y-X, new1(M,N,X,Y1,Z).`
9. `new1(M,N,X,Y,Z) :- X=Y,Z=X, gcd(M,N,D),Z≠D.`

Clauses 6 and 9 can be rewritten, respectively, as:

10. `incorrect :- M≥1,N≥1,Z≠D, new1(M,N,M,N,Z), gcd(M,N,D).`
11. `new1(M,N,X,Y,Z) :- X=Y, Z=X.`

This rewriting is correct because `new1` modifies the values of neither `M` nor `N`.

Note that we could avoid performing the above rewriting and automatically derive a similar program where the constraints characterizing the initial and the error properties occur in the same clause, by starting our derivation from a more general definition of the reachability relation. However, an in-depth analysis of

[1]The atom `gcd(X,Y,D)` is considered to be not unfoldable.

this variant of our verification method is beyond the scope of this thesis (see also [119] for a discussion about different styles of encoding the reachability relation and the semantics of imperative languages in CLP).

Now we perform the Verification Conditions Transformation step of the verification method by applying the *Transform*$_{prop}$ strategy to the program consisting of the following clauses: 1, 2, 3, 7, 8, 10, and 11.

Unfolding. We start off by unfolding clause 10 with respect to the atom `new1(M,N,M,N,Z)`, and we get:

12. `incorrect :- M≥1, N≥1, M>N, X1=M−N, Z≠D,`
    `new1(M,N,X1,N,Z), gcd(M,N,D).`
13. `incorrect :- M≥1, N≥1, M<N, Y1=N−M, Z≠D,`
    `new1(M,N,M,Y1,Z), gcd(M,N,D).`
14. `incorrect :- M≥1, N≥1, M=N, Z=M, Z≠D, gcd(M,N,D).`

By unfolding clauses 12–14 with respect to the atom `gcd(M,N,D)` we derive:

15. `incorrect :- M≥1, N≥1, M>N, X1=M−N, Z≠D,`
    `new1(M,N,X1,N,Z), gcd(X1,N,D).`
16. `incorrect :- M≥1, N≥1, M<N, Y1=N−M, Z≠D,`
    `new1(M,N,M,Y1,Z), gcd(M,Y1,D).`

(Note that by unfolding clause 14 we get an empty set of clauses because the constraints derived in this step are all unsatisfiable.) Sine no unfoldable atom occurs in the body of clauses 15 and 15 the Unfolding phase terminates. Indeed, according to the Unfolding phase presented in Figure 13, we have that: (i) the atoms `new1(M,N,X1,N,Z)` and `gcd(X1,N,D)` in clause 15, and (ii) the atoms `new1(M,N,M,Y1,Z)` and `gcd(M,Y1,D)` in clause 16, have been derived by unfolding atoms with the same predicate symbols.

The Constraint Replacement and Clause Removal phases do not modify the set of clauses derived after the Unfolding phase because no laws are available for the predicate `gcd`.

Definition Introduction. In order to fold clauses 15 and 16, we perform a generalization step and we introduce a new predicate defined by the following clause:

17. `new2(M,N,X,Y,Z,D) :- M≥1, N≥1, Z≠D, new1(M,N,X,Y,Z), gcd(X,Y,D).`

The body of this clause is the most specific generalization of the bodies of clauses 10, 15 and 16. Here, we define a conjunction `G` to be a generalization of a conjunction `C` if there exists a substitution ϑ such that `G`ϑ can be obtained by deleting some of the conjuncts of `C`.

Clause 17 defining the new predicate `new2`, is added to *Defs* and *InDefs* and, since the latter set is not empty, we perform a new iteration of the while-loop body of the $\mathit{Transform}_{prop}$ strategy.

UNFOLDING. By unfolding clause 17 w.r.t. `new1(M,N,X,Y,Z)` and then unfolding the resulting clauses w.r.t. `gcd(X,Y,Z)`, we derive:

```
18. new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X>Y, X1=X−Y, Z≠D,
      new1(M,N,X1,Y,Z), gcd(X1,Y,D).
19. new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X<Y, Y1=Y−X, Z≠D,
      new1(M,N,X,Y1,Z), gcd(X,Y1,D).
```

Since all the atoms occurring in clauses 18 and 19 have been derived by unfolding atoms with the same predicate symbols, the UNFOLDING phase terminates.

DEFINITION INTRODUCTION. Clauses 18 and 19 can be folded by using clause 17, thus, there is no need to introduce any new definition. Since no clause to be processed is left the $\mathit{Transform}_{prop}$ exits the outermost while-loop.

FOLDING. Finally, we get the following final program `S` by folding clauses 15, 16, 18, and 19 using the definition clause 17.

```
20. incorrect :-M≥1,N≥1,M>N,X1=M−N,Z≠D, new2(M,N,X1,N,Z,D).
21. incorrect :-M≥1,N≥1,M<N,Y1=N−M,Z≠D, new2(M,N,M,Y1,Z,D).
22. new2(M,N,X,Y,Z,D) :-M≥1,N≥1, X>Y,X1=X−Y,Z≠D,
      new2(M,N,X1,Y,Z).
23. new2(M,N,X,Y,Z,D) :-M≥1, N≥1,X<Y,Y1=Y−X,Z≠D,
      new2(M,N,X,Y1,Z).
```

The final program consisting of clause 20, 21, 22, and 23 contains no constrained facts. Hence both predicates `incorrect` and `new2` are useless and all clauses of `S` can be deleted. Thus, the Verification Conditions Analysis trivially terminates by producing the answer '*correct*' and we conclude that the imperative program *gcd* is correct with respect to the given properties φ_{init} and φ_{error}. □

The application of the powerful transformation rules we have considered in this chapter enables the verification of a larger class of properties, but it does not entirely fit into the automated strategy used in Chapter 5. We are currently considering the issue of designing fully mechanizable strategies for guiding the application of our program transformation rules.

The rule-based program transformation technique presented here is related to *conjunctive partial deduction* (CPD) [48], a technique for the specialization of logic programs with respect to conjunctions of atoms. There are, however, some substantial differences between CPD and the approach we have presented here.

First, CPD is not able to specialize logic programs with constraints and, thus, it cannot be used to prove the correctness of the `GCD` program where the role of constraints is crucial. Indeed, using the ECCE conjunctive partial deduction system [100] for specializing the program consisting of clauses {1,2,3,7,8,10,11} with respect to the query `incorrect`, we obtain a residual program where the predicate `incorrect` is not useless. Thus, we cannot conclude that the atom `incorrect` does not belong to the least model of the program, and thus we cannot conclude that the program is correct. One more difference between CPD and our technique is that we may use constraint replacement rules which allow us to evaluate terms over domain-specific theories. In particular, we can apply the goal replacement rules using well-developed theories for data structures like arrays, lists, heaps and sets (see [20, 110, 70, 14, 126, 140] for some formalizations of these theories).

CHAPTER 8

The VeriMAP Software Model Checker

In this chapter we present our tool for program verification, called VeriMAP, which is based on the *transformation* techniques for CLP programs presented in Chapters 3–6.

The current version of VeriMAP can be used for verifying partial correctness properties of C programs that manipulate integers and arrays.

From the CLP representation of the given C program and the initial and error properties, VeriMAP generates a set of verification conditions (VC's) in the form of CLP clauses. The VC generation is performed by a transformation that consists in specializing (with respect to the given C program and the initial and error properties) a CLP program that defines the operational semantics of the C language and the proof rules for verifying partial correctness (see Chapter 3). Then, the CLP program made out of the generated VC's is transformed by applying unfold/fold transformation rules. This transformation 'propagates' the constraints occurring in the CLP clauses and derives equisatisfiable, easier to analyze VC's (see Chapters 4 and 5). During constraint propagation VeriMAP makes use of constraint solvers for linear (integer or rational) arithmetic and array formulas (see Chapter 6). In a subsequent phase the transformed VC's are processed by a lightweight analyzer that basically consists in a bounded unfolding of the clauses. Since partial correctness is in general undecidable, the analyzer may not be able to detect the satisfiability or the unsatisfiability of the VC's and, if this is the case, the verification process continues by iterating the transformation and the propagation of the constraints in the VC's (see Chapter 5).

This chapter is structured as follows. In Section 8.1 we present the architecture of the software model checker. Then, in Section 8.2 we show through some

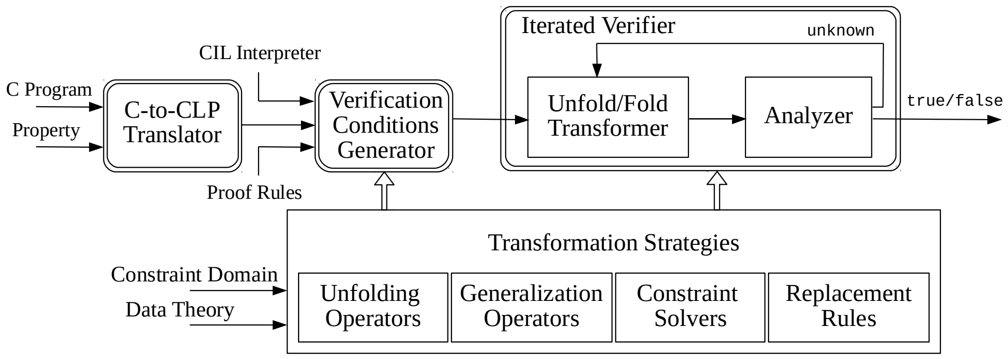


Figure 14: The VeriMAP architecture.

examples how to use the VeriMAP tool.

8.1 Architecture

The VeriMAP tool consists of three modules (see Figure 14). (1) A *C-to-CLP Translator* (*C2CLP*) that constructs a CLP encoding of the C program and of the property given as input (that is, it performs the CLP Translation step). (2) A *Verification Conditions Generator* (*VCG*) that generates a CLP program representing the VC's for the given program and property (that is, it performs the Verification Conditions Generation step). The *VCG* module takes as input also the CLP representations of the operational semantics of CIL and of the proof rules for establishing partial correctness. (3) An *Iterated Verifier* (*IV*) that attempts to determine whether or not the VC's are satisfiable by iteratively applying unfold/fold transformations to the input VC's, and analyzing the derived VC's (that is, it performs the Verification Conditions Transformation and Verification Conditions Analysis steps).

The *C2CLP* module is based on a modified version of the CIL tool [115]. This module first parses and type-checks the input C program, annotated with the property to be verified, and then transforms it into an equivalent program written in CIL that uses a reduced set of language constructs. During this transformation, in particular, commands that use `while`'s and `for`'s are translated into equivalent commands that use `if-then-else`'s and `goto`'s. This transformation step simplifies the subsequent processing steps. Finally, *C2CLP* generates as output the CLP encoding of the program and of the property by running a custom implementation of the CIL visitor pattern [115]. In particular, for each program command, *C2CLP* generates a CLP fact of the form `at(L, C)`, where `C` and `L` represent the command and its label, respectively. *C2CLP* also constructs the clauses for the predicates `phiInit` and `phiError` representing the formulas φ_{init} and φ_{error} that specify the property.

The *VCG* module generates the VC's for the given program and property by applying a program specialization technique based on equivalence preserving unfold/fold transformations of CLP programs [60]. Similarly to what has been proposed in [119], the *VCG* module specializes the interpreter and the proof rules with respect to the CLP representation of the program and the property generated by *C2CLP* (that is, the clauses defining `at`, `phiInit`, and `phiError`). The output of the specialization process is the CLP representation of the VC's. This specialization process is said to 'remove the interpreter' in the sense that it removes every reference to the predicates used in the CLP definition of the interpreter in favour of new predicates corresponding to (a subset of) the 'program points' of the original C program. Indeed, the structure of the call-graph of the CLP program generated by the *VCG* module corresponds to that of the control-flow graph of the C program.

The *IV* module consists of two submodules: (i) the *Unfold/Fold Transformer*, and (ii) the *Analyzer*. The *Unfold/Fold Transformer* propagates the constraints occurring in the definition of `phiInit` and `phiError` through the input VC's thereby deriving a new, equisatisfiable set of VC's. The *Analyzer* checks the satisfiability of the VC's by performing a lightweight analysis. The output of this analysis is either (i) `true`, if the VC's are satisfiable, and hence the program is correct, or (ii) `false`, if the VC's are unsatisfiable, and hence the program is incorrect (and a counterexample may be extracted), or (iii) `unknown`, if the lightweight analysis is unable to determine whether or not the VC's are satisfiable. In this last case the verification continues by iterating the propagation of constraints by invoking again the *Unfold/Fold Transformer* submodule. At each iteration, the *IV* module can also apply a *Reversal* transformation [42], with the effect of reversing the direction of the constraint propagation (either from `phiInit` to `phiError` or vice versa, from `phiError` to `phiInit`).

The *VCG* and *IV* modules are realized by using MAP [108], a transformation engine for CLP programs (written in SICStus Prolog), with suitable concrete versions of *Transformation Strategies*. There are various versions of the transformation strategies which, as indicated in [42], are defined in terms of: (i) *Unfolding Operators*, which guide the symbolic evaluation of the VC's, by controlling the expansion of the symbolic execution trees, (ii) *Generalization Operators* [62], which guarantee termination of the *Unfold/Fold Transformer* and are used (together with widening and convex-hull operations) for the automatic discovery loop invariants, (iii) *Constraint Solvers*, which check satisfiability and entailment within the Constraint Domain at hand (for example, the integers or the rationals), and (iv) *Replacement Rules*, which guide the application of the axioms and the properties of the Data Theory under consideration (like, for example, the theory of arrays), and their interaction with the Constraint Domain.

8.2 Usage

VeriMAP can be downloaded from `http://map.uniroma2.it/VeriMAP` and can be run by executing the following command:

```
./VeriMAP program.c
```

where `program.c` is the C program annotated with the property to be verified. VeriMAP has options for applying custom transformation strategies and for exiting after the execution of the *C2CLP* or *VCG* modules, or after the execution of a given number of iterations of the *IV* module.

8.3 Proving Partial Correctness with the VeriMAP Tool

In this section we show how to use VeriMAP on two simple examples. Let us consider the C Program listed below.

```c
int x, y;
void sum_and_set(int z) {
  x = x+y;
  y = z;
}
int main() {
  if (x < y) {
    sum_and_set(0);
    if (x > y)
      goto END;
  }
  while (x <= y)
    x=x+1;
END:
  return 0;
}
```

The property considered in this example is defined by the formulas $\varphi_{init} \equiv y \geq 0$ and $\varphi_{error} \equiv x \leq 0$, which are encoded as follows within a C comment.

```
/*
  % MAP_specification
  phiInit(G) :- lookup(scalar(int(y)),G,Y), Y>=0.
  phiError(G) :- lookup(scalar(int(x)),G,X), X=<0.
*/
```

The argument `G` of `phiInit` and `phiError` is a list of pairs binding the global variables of the C program to the corresponding CLP variables. This binding is obtained by using the `lookup` predicate which is compiled away during the Verification Conditions Generation step.

In order to be compliant with the rules of the TACAS Verification Competition [12], a user may also encodes initial and error properties the program as calls to the functions `__VERIFIER_assume` and `__VERIFIER_assert`, thereby producing the following C program.

```
int x, y;
void sum_and_set(int z) {
  x = x+y;
  y = z;
}
void main() {
  __VERIFIER_assume(y >= 0);
  if (x < y) {
    sum_and_set(0);
    if (x > y)
      goto END;
  }
  while (x <= y)
    x=x+1;
END:
  __VERIFIER_assert(x > 0);
}
```

We assume that the program to be verified, together with the property, is stored in a file named `example1.c`.

By executing `./VeriMAP example1.c` we get as output: `Answer: true`, which means that the given program is correct. In the examples presented in this section, the *Unfold/Fold Transformer* module makes use of a generalization operator based on standard widening.

In the following we will see how this answer is produced. In particular, we will list the commands needed to invoke each module described in Section 8.2 and the output produced by each module.

C-to-CLP Translation

The *C2CLP* module is invoked by the command `./VeriMAP --c2clp example1.c`. The output is the following set of CLP clauses (`example1.pl`), which define the predicates `fun/4`, `at/2` and `gvars/1` representing the function declarations, the C statements, and the global variable declarations, respectively.

```
% function declarations
fun(sum_and_set,[id(loc(scalar(int(z))))],[],entry_point(addr(9))).
fun(main,[],[id(loc(scalar(int(x))))],entry_point(addr(11)).
% function definitions
at(lab(9,inst),asgn(id(glb(scalar(int(x)))),
    aexp(plus(aexp(id(glb(scalar(int(x))))),
    aexp(id(glb(scalar(int(y))))))),addr(9.1))).
at(lab(9.1,inst),asgn(id(glb(scalar(int(y)))),
    aexp(const(int(0))),addr(10))).
at(lab(10,ret),ret(aexp(id(loc(scalar(int(w))))))).
at(lab(11,ifte),ite(bexp(lt(aexp(id(glb(scalar(int(x))))),
    aexp(id(glb(scalar(int(y))))))),addr(12),addr(17))).
at(lab(12,inst),call(sum_and_set,[aexp(id(glb(scalar(int(x)))))],
    id(undef),addr(13))).
at(lab(13,ifte),ite(bexp(gt(aexp(id(glb(scalar(int(x))))),
    aexp(id(glb(scalar(int(y))))))),addr(14),addr(17))).
at(lab(14,goto),goto(addr(22))).
at(lab(17,goto),goto(addr(18))).
at(lab(18,loop),ite(bexp(lte(aexp(id(glb(scalar(int(x))))),
    aexp(id(glb(scalar(int(y))))))),addr(20),addr(19))).
at(lab(19,goto),goto(addr(22))).
at(lab(20,inst),asgn(id(glb(scalar(int(x)))),
    aexp(plus(aexp(id(glb(scalar(int(x))))),
    aexp(const(int(1))))),addr(17))).
```

```
25  at(lab(22,ret),ret(aexp(const(int(0))))).
26  at(lab(h,halt),halt).
27  % global variables
28  gvars([
29      (id(glb(scalar(int(x)))),aexp(id(undef))),
30      (id(glb(scalar(int(y)))),aexp(id(undef)))
31     ]).
```

The predicate `fun(Id,Parameters,LocalVars,EntryAddr)` represents the function definitions, where: (i) `Id` is the identifier of the function, (ii) `Parameters` is the list of the formal parameters, (iii) `LocalVars` is the list of the local variables, and (iv) `EntryAddr` is the address of the first command in the body of the function. For instance, lines 5–10 represent:

`int sum_and_set(int z) { x = x+y; y = 0; }`.

The predicate `at(Lab,Cmd)` represents a C command, where: (i) `Lab` is of the form `lab(Addr,Type)` and represents the label of the command (in particular, `Addr` and `Type` represent the address of the entry point and the command type, respectively), and (ii) `Cmd` represents the given C command. For instance, lines 11–12 represent the conditional command at lines 7–11 of the given C program. The first argument of `ite` represents the expression of the command, where: (i) `lt` represents the '$<$' operator, (ii) `bexp` and `aexp` represent boolean and arithmetic expressions, respectively, and (iii) `loc` and `glb` represent local and global variable identifiers, respectively. The second and third arguments of `ite` represent the address of the first instruction of the conditional branches.

The predicate `gvars(GlbList)` represents the list of global variables. In the example we have a two global variables `id(glb(scalar(int(y))))` and `id(glb(scalar(int(x))))`, which are uninitialized (see `aexp(id(undef))`).

Verification Conditions Generation

By executing the command `./VeriMAP --vcg example1.c` the verification process stops after the execution of the *VCG* module. This module specializes the following proof rules for proving partial correctness:

```
correct :- \+ incorrect.
incorrect :- elem(X,init), reach(X,U).
reach(X,U) :- elem(X,error).
reach(X,U) :- tr(X,Y), reach(Y,U).
elem(X,init) :- phiInit(X).
```

```
 6  elem(X,error) :- phiError(X).
```

where \+ denotes negation, `tr` denotes the transition relation that defines the *CIL Interpreter*, `elem(X,init)` and `elem(X,error)` denote the properties that characterize the initial and error configurations, respectively. Thus, `correct` holds if and only if there exists no error configuration which is reachable from some initial configuration. The predicate `tr/2` is defined by the CLP clauses presented in Section 3.2.

The generation of the VC's is performed by specializing the proof rules and the interpreter with respect to the set of CLP clauses produced by applying *C2CLP* to `example1.c`, that is, the clauses for `at`, `phiInit`, and `phiError`. In the following we present an excerpt of the log file produced by invoking the *VCG* module.

```
 7  [...]         <--- missing lines here
 8  Transformed program:
 9  new7(A,B) :- A-B=<0, new5(A,B).
10  new5(A,B) :- C=1+A, A-B=<0, new5(C,B).
11  new4(A,B) :- C=A+B, D=0, new7(C,D).
12  new3(A,B) :- A-B=< -1, new4(A,B).
13  new3(A,B) :- A-B>=0, new5(A,B).
14  new2(A,B) :- new3(A,B).
15  new5(A,B) :- A=<0, A-B>=1.
16  new7(A,B) :- A=<0, A-B>=1.
17  incorrect :- A>=0, new2(B,A).
18  correct :- \+incorrect.
```

Unfold/Fold Transformation

By executing `./VeriMAP --transform example1.c` the verification process terminates after one execution of the *Unfold/Fold Transformer* module.

This module transforms the VC's generated as output by the *VCG* module. In order to maximize code reuse, the VC's are first converted into a transition relation representation.

The excerpt of the log file below shows the part of the transition relation `tr` (lines 20–25) corresponding to the clauses listed at lines 9–14, and the three `elem` facts (line 26–28) corresponding to the clauses at lines 15–17, respectively. Lines 31–40 list the transformed program, and lines 42–44 give some statistics about the transformation (in particular, the number of Unfold-Definition-Fold cycles, the number of clauses introduced by the definition rule, and the time required by the transformation process).

```
19  Initial program:
20  tr(s(new7,A,B),s(new5,A,B)) :- A-B=<0.
21  tr(s(new5,A,B),s(new5,C,B)) :- C=1+A, A-B=<0.
22  tr(s(new4,A,B),s(new7,C,D)) :- C=A+B, D=0.
23  tr(s(new3,A,B),s(new4,A,B)) :- A-B=< -1.
24  tr(s(new3,A,B),s(new5,A,B)) :- A-B>=0.
25  tr(s(new2,A,B),s(new3,A,B)).
26  elem(s(new5,A,B),error) :- A=<0, A-B>=1.
27  elem(s(new7,A,B),error) :- A=<0, A-B>=1.
28  elem(s(new2,A,B),initial) :- B>=0.
29
30  Transformed program:
31  new9(A,B) :- A= -1+C, B=0, C=<1, new9(C,B).
32  new8(A,B) :- A= -1+C, B=0, C=<1, new9(C,B).
33  new7(A,B) :- B=0, A=<0, new8(A,B).
34  new5(A,B) :- A= -1+C, B= -1+C, C>=1, new5(C,B).
35  new4(A,B) :- A=-(B)+C, D=0, B>=0, B-1/2*C>=1/2, new7(C,D).
36  new3(A,B) :- A-B=< -1, B>=0, new4(A,B).
37  new3(A,B) :- B>=0, A-B>=0, new5(A,B).
38  new2(A,B) :- B>=0, new3(A,B).
39  incorrect :- A>=0, new2(B,A).
40  correct :- \+incorrect.
41
42  #UDF-iteration(s): 9
43  #definitions: 9
44  Elapsed time 10ms
```

Analysis

As a last step, the `./VeriMAP example1.c` command invokes the *Analyzer* module which detects the absence of facts in the transformed CLP program (lines 31–40). Thus, no unfolding will ever derive a fact for the predicate `incorrect`, and hence the predicate `correct` is true. The *Analyzer* module produces the output `Answer: true`, meaning that the program in `example1.c` is correct.

Iterated Verification

Now we consider a second C program (in file `example2.c`)

```
int x=0, y=0, n;
while (x < n) {
  x = x+1;
  y = y+x;
}
__VERIFIER_assert( x<=y );
```

By executing the command `./VeriMAP example2.c` we get: `Answer:  unknown`. Indeed, at the end of the process we derive the following program:

```
 1  new10(A,B,C,D) :- D=0, B>=0, A-B>=1, A-C>=0.
 2  new8(A,B,C) :- D=1, A-B=<0, A>=0, A-C>=0, new9(A,B,C,D).
 3  new8(A,B,C) :- D=0, B>=0, A-B>=1, A-C>=0, new10(A,B,C,D).
 4  new5(A,B,C) :- A=0, B=0, D=1, C=<0, new6(A,B,C,D).
 5  new4(A,B,C) :- A= -1+D, B=E-D, D>=1, C-D>=0, E-D>=0,
        new4(D,E,C).
 6  new4(A,B,C) :- A>=0, B>=0, A-C>=0, new8(A,B,C).
 7  [...]          <--- missing lines here
 8  new2(A,B,C) :- A=0, B=0, new3(A,B,C).
 9  incorrect :- A=0, B=0, new2(A,B,C).
10  correct :- \+incorrect.
```

where the presence of the constrained fact at line 1 allows the lightweight analyzer to give neither the answer `true` nor the answer `false`. Thus, the *IV* module performs one more invocation of the transformation and analysis submodules. (The first step of the *Unfold/Fold Transformer* is an application of the *Reversal* transformation to enable the propagation of the constraints occurring in the definition of `phiError`).

The excerpt of the log file reported below shows some information about the transformation performed at the second iteration of *IV*.

```
11  Initial program:
12  tr(s(new2,A,B,C),s(new3,A,B,C)) :- A=0, B=0.
13  [...]          <--- missing lines here
14  tr(s(new4,A,B,C),s(new4,D,E,C)) :-
15     A= -1+D, B=E-D, D>=1, C-D>=0, E-D>=0.
16  tr(s(new4,A,B,C),s(new8,A,B,C)) :-
17     A>=0, B>=0, A-C>=0.
```

```
tr(s(new5,A,B,C),s(new6,A,B,C,D)) :-
   A=0, B=0, D=1, C=<0.
tr(s(new8,A,B,C),s(new9,A,B,C,D)) :-
   D=1, A-B=<0, A>=0, A-C>=0.
tr(s(new8,A,B,C),s(new10,A,B,C,D)) :-
   D=0, B>=0, A-B>=1, A-C>=0.
elem(s(new2,A,B,C),initial) :-
   A=0, B=0.
elem(s(new10,A,B,C,D),error) :-
   D=0, B>=0, A-B>=1, A-C>=0.

Transformed program:
new3(A,B,C) :- B>=0, A-B>=1, A-C>=0, new4(A,B,C).
new2(A,B,C,D) :- D=0, B>=0, A-B>=1, A-C>=0, new3(A,B,C).
incorrect :- A=0, B>=0, C-D>=0, C-B>=1, new2(C,B,D,A).
correct:- \+incorrect.

#definitions: 5
#UDF-iteration(s): 5
Elapsed time 10ms
```

Since the transformed CLP program contains no constrained facts, the *Analyzer* module concludes that the program of `example2.c` is correct and returns `Answer: true`.

The iterated verification shown here has been performed by executing the command `./VeriMAP --iterations=2 example2.pl` (where '`--iterations=2`' specifies that the maximal number of iterations to be executed is 2).

PART II

SYNTHESIS

CHAPTER 1

Synthesizing Concurrent Programs using Answer Set Programming

We consider *concurrent programs* consisting of finite sets of *processes* which interact with each other by using a shared variable ranging over a finite domain. The interaction protocol is realized in a distributed manner, that is, every process includes some instructions which operate on the shared variable.

Even for a small number of processes, interaction protocols which guarantee a desired behaviour of the concurrent programs may be hard to design and prove correct. Thus, people have been looking for methods for the automatic synthesis of concurrent programs from the formal specification of their behaviour. Among those methods we recall the ones proposed by Clarke and Emerson [25], Manna and Wolper [107], and Attie and Emerson [7, 8], which use tableau-based algorithms, and those proposed by Pnueli and Rosner [122], and Kupferman and Vardi [96], which use automata-based algorithms.

In contrast with those approaches we do not present an *ad-hoc* algorithm for synthesizing concurrent programs and, instead, we propose a framework based on logic programming by which we reduce the problem of synthesizing concurrent programs to the problem of computing models of a logic program encoding a given specification. We assume that *behavioural properties* of concurrent programs, such as safety or liveness properties, are specified by using formulas of the Computation Tree Logic (CTL), which is a very popular propositional temporal logic over branching time structures [25, 27]. This temporal, behavioural specification φ is encoded as a logic program Π_φ. We also assume that the processes to be synthesized satisfy suitable *structural properties*, such as *symmetry* properties, which specify that all processes follow the same cycling pattern of possible actions. Such structural properties cannot be easily specified by using CTL formulas and, in order to overcome this difficulty, we use, instead, a simple algebraic structure which can be specified in predicate logic and are encoded as a logic program Π_σ. Thus, the specification of a concurrent program to be synthesized consists of a logic program $\Pi = \Pi_\varphi \cup \Pi_\sigma$ which encodes both the behavioural and the structural properties that the concurrent program should enjoy.

In order to construct models of the program Π, we use logic programming with the answer set semantics and we show that every answer set of Π encodes a concurrent program satisfying the given specification. Thus, by using an Answer Set Programming (ASP) solver, such as the ones presented in [66, 53, 71, 89, 99, 134], which computes the answer sets of logic programs, we can synthesize concurrent programs which enjoy the desired behavioural and structural properties. We have performed some synthesis experiments (see Chapter 4) and, in particular, we have synthesized some protocols which are guaranteed to enjoy behavioural properties such as mutual exclusion, starvation freedom, and bounded overtaking, and also suitable symmetry properties. However, the synthesis framework we propose is general and it can be applied to many other classes of concurrent systems and properties besides those mentioned above.

The remaining part of this chapter is devoted to recall some basic notions and the terminology we will use.

1.1 Preliminaries

In this section we will present: (i) the syntax of (a variant of) the *guarded commands* [51], which we use for defining concurrent programs, (ii) some basic notions of group theory, which are required for defining the so-called symmetric concurrent programs, and (iii) some fundamental concepts of Computation Tree Logic and of Answer Set Programming, which we use for our synthesis method.

1.1.1 Guarded commands

The guarded commands we consider are defined from the following two basic sets: (i) variables, v in *Var*, each ranging over a finite domain D_v, and (ii) guards, g in *Guard*, of the form: $g ::= \mathit{true} \mid \mathit{false} \mid v = d \mid \neg\, g \mid g_1 \wedge g_2$, with $v \in \mathit{Var}$ and $d \in D_v$. We also have the following derived sets whose definitions are mutually recursive: (iii) commands, c in *Command*, of the form:

$c ::= \mathit{skip} \mid v := d \mid c_1 \,;\, c_2 \mid$ `if` gc `fi` $\mid$ `do` gc `od`, where '$;$' denotes the *sequential composition* of commands which is associative, and (iv) guarded commands, gc in *GCommand*, of the form: $gc ::= g \to c \mid gc_1 \;[\!]\; gc_2$, where '$[\!]$' denotes the *parallel composition* of guarded commands which is associative and commutative.

The operational semantics of commands can be described in an informal way as follows. *skip* does nothing. $v := d$ stores the value d in the location of the variable v. In order to execute $c_1; c_2$ the command c_1 is executed first, and if c_1 does not fail (see below) then the command c_2 is executed. In order to execute `if` $gc_1 \;[\!]\; \ldots \;[\!]\; gc_n$ `fi`, with $n \geq 1$, one of the guarded commands $g \to c$ in $\{gc_1, \ldots, gc_n\}$ whose guard g evaluates to *true*, is non-deterministically chosen, and then c is executed; otherwise, if no guard of a guarded command in $\{gc_1, \ldots, gc_n\}$ evaluates to *true*, then the whole command `if ... fi` terminates with failure. In order to execute `do` $gc_1 \;[\!]\; \ldots \;[\!]\; gc_n$ `od`, with $n \geq 1$, one of the guarded commands $g \to c$ in $\{gc_1, \ldots, gc_n\}$ whose guard g evaluates to *true*, is non-deterministically chosen, then c is executed and the whole command `do`...`od` is executed again; otherwise, if no guard of a guarded command in $\{gc_1, \ldots, gc_n\}$ evaluates to *true*, then the execution proceeds with the next command. The formal semantics of commands will be given in the next section.

1.1.2 Groups

A group G is a pair $\langle S, \circ \rangle$, where S is a set and $\circ$ is a binary operation on S satisfying the following axioms: (i) $\forall x, y \in S.\, x \circ y \in S$, (ii) $\forall x, y, z \in S.\, (x \circ y) \circ z = x \circ (y \circ z)$, (iii) $\exists e \in S.\, \forall x \in S.\; e \circ x = x \circ e = x$, and (iv) $\forall x \in S.\, \exists y \in S.\; x \circ y = y \circ x = e$. The element e is the *identity* of the group G and the cardinality of S is the *order of the group* G. For any $x \in S$, for any $n \geq 0$, we write x^n to denote the term $x \circ \ldots \circ x$ with n occurrences of x. We stipulate that x^0 is e.

A group $G = \langle S, \circ \rangle$ is said to be *cyclic* iff there exists an element $x \in S$, called a *generator*, such that $S = \{x^n \mid n \geq 0\}$. We denote by *Perm*$(S)$ the set of all permutations on the set S, that is, the set of all bijections from S to S. *Perm*(S) is a group whose operation $\circ$ is function composition and the identity e is the identity permutation, denoted *id*. Given a finite set S, the *order of a permutation* p in *Perm*(S) is the smallest natural number n such that $p^n = \mathit{id}$.

1.1.3 Computation Tree Logic

Computation Tree Logic (CTL) is a propositional branching time temporal logic [27]. The underling time structure is a *tree of states.* Every state denotes an instant in time and may have many successor states. There are quantifiers over paths of the tree: A (*for all paths*) and E (*for some path*), which are used for specifying properties that hold for all paths or for some path, respectively. Together with these quantifiers, there are temporal operators such as: X (*next state*), F (*eventually*), G (*globally*), and U (*until*), which are used for specifying properties that hold in the states along paths of the tree. Their formal semantics will be given below.

Given a finite nonempty set *Elem* of elementary propositions ranged over by p, the syntax of CTL formulas φ is as follows:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{E}[\varphi_1\,\mathsf{U}\,\varphi_2]$$

We introduce the following abbreviations:

(*i*) *true* for $\varphi\vee\neg\varphi$, where φ is any CTL formula,

(*ii*) *false* for $\neg$*true*,

(*iii*) $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1\wedge\neg\varphi_2)$,

(*iv*) $\mathsf{EF}\varphi$ for $\mathsf{E}[\mathit{true}\,\mathsf{U}\,\varphi]$

(*v*) $\mathsf{AG}\,\varphi$ for $\neg\mathsf{EF}\,\neg\varphi$,

(*vi*) $\mathsf{AF}\,\varphi$ for $\neg\mathsf{EG}\,\neg\varphi$,

(*vii*) $\mathsf{A}[\,\varphi_1\mathsf{U}\varphi_2]$ for $\neg\mathsf{E}[\neg\varphi_2\,\mathsf{U}\,(\neg\varphi_1\wedge\neg\varphi_2)]\wedge\mathsf{AF}\,\varphi_2$, and

(*viii*) $\mathsf{AX}\,\varphi$ for $\neg\mathsf{EX}\,\neg\,\varphi$.

The semantics of CTL is provided by a Kripke structure $\mathcal{K}=\langle\mathcal{S},\mathcal{S}_0,\mathcal{R},\lambda\rangle$, where: (i) $\mathcal{S}$ is a finite set of *states*, (ii) $\mathcal{S}_0\subseteq\mathcal{S}$ is a set of *initial states*, (iii) $\mathcal{R}\subseteq\mathcal{S}\times\mathcal{S}$ is a *total transition relation* (thus, $\forall u\in\mathcal{S}.\ \exists v\in\mathcal{S}.\ \langle u,v\rangle\in\mathcal{R}$), and (iv) $\lambda:\mathcal{S}\to\mathcal{P}(\mathit{Elem})$ is a *total labelling function* that assigns to every state $s\in\mathcal{S}$ a subset $\lambda(s)$ of the set *Elem*. A path π in $\mathcal{K}$ from a state s_0 is an infinite sequence $\langle s_0,s_1,\ldots\rangle$ of states such that, for all $i\geq 0$, $\langle s_i,s_{i+1}\rangle\in\mathcal{R}$. The fact that a CTL formula φ holds in a state s of a Kripke structure $\mathcal{K}$ will be denoted by $\mathcal{K},s\vDash\varphi$. For any CTL formula φ and state s, we define the relation $\mathcal{K},s\vDash\varphi$ as follows:

$\mathcal{K},s \vDash p$	iff	$p \in \lambda(s)$
$\mathcal{K},s \vDash \neg\,\varphi$	iff	$\mathcal{K},s \vDash \varphi$ does not hold
$\mathcal{K},s \vDash \varphi_1 \wedge \varphi_2$	iff	$\mathcal{K},s \vDash \varphi_1$ and $\mathcal{K},s \vDash \varphi_2$
$\mathcal{K},s \vDash \mathsf{EX}\,\varphi$	iff	there exists $\langle s,t\rangle \in \mathcal{R}$ such that $\mathcal{K},t \vDash \varphi$
$\mathcal{K},s \vDash \mathsf{E}[\varphi_1 \mathsf{U}\,\varphi_2]$	iff	there exists a path $\langle s_0,s_1,s_2,\ldots\rangle$ in $\mathcal{K}$ with $s_0 = s$ such that for some $i \geq 0$, $\mathcal{K}, s_i \vDash \varphi_2$ and for all $0 \leq j < i$, $\mathcal{K}, s_j \vDash \varphi_1$
$\mathcal{K},s \vDash \mathsf{EG}\,\varphi$	iff	there exists a path $\langle s_0,s_1,s_2,\ldots\rangle$ in $\mathcal{K}$ with $s_0 = s$ such that for all $i \geq 0$, $\mathcal{K}, s_i \vDash \varphi$.

Thus, in particular we have that: (i) $\mathcal{K},s \vDash \mathsf{EX}\,\varphi$ holds iff in $\mathcal{K}$ there exists a successor of state s which satisfies φ, (ii) $\mathcal{K},s \vDash \mathsf{E}[\varphi_1 \mathsf{U}\,\varphi_2]$ holds iff there exists a path in $\mathcal{K}$ starting at s along which there exists a state where φ_2 holds and φ_1 holds in every preceding state, and (iii) $\mathcal{K},s \vDash \mathsf{EG}\,\varphi$ holds iff in $\mathcal{K}$ there exists a path starting at s where φ holds in every state along that path.

1.1.4 Answer Set Programming

Answer set programming (ASP) is a declarative programming paradigm based on logic programs and their answer set semantics. Now we recall some basic definitions of ASP and for those not recalled here the reader may refer to [10, 19, 54, 68, 69, 136]. A *term* t is either a variable X or a function symbol f of arity n (≥ 0) applied to n terms $f(t_1, \ldots, t_n)$. If $n = 0$ then f is called a *constant*. An *atom* is a predicate symbol p of arity n (≥ 0) applied to n terms $p(t_1, \ldots, t_n)$. A *rule* is an implication of the form:

$$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1} \wedge \ldots \wedge a_m \wedge \texttt{not}\, a_{m+1} \wedge \ldots \wedge \texttt{not}\, a_n$$

where $a_1, \ldots, a_k, a_{k+1}, \ldots, a_n$ (for $k \geq 0$, $n \geq k$) are atoms and '`not`' denotes negation as failure. A rule with $k > 1$ is said to be a *disjunctive rule* and each atom in $\{a_1, \ldots, a_k\}$ is called a *disjunct*. A rule with $k = 1$ is called *normal*. A rule with $k = 0$ is called an *integrity constraint*. A rule with $k = n$ is called a *fact*. A *logic program* Π is a set of rules. It is said to be a *disjunctive logic program* if there exists a disjunctive rule and it is said to be a *normal logic program* if for every rule $k \leq 1$.

Given a rule r, we define the following sets: $H(r) = \{a_1, \ldots, a_k\}$, $B^+(r) = \{a_{k+1}, \ldots, a_m\}$, $B^-(r) = \{a_{m+1}, \ldots, a_n\}$, and $B(r) = B^+(r) \cup B^-(r)$ and we introduce the following abbreviations: $\mathit{head}(r) = \bigvee_{a \in H(r)} a$, $\mathit{pos}(r) = \bigwedge_{a \in B^+(r)} a$, $\mathit{neg}(r) = \bigwedge_{a \in B^-(r)} \texttt{not}\ a$, and $\mathit{body}(r) = \mathit{pos}(r) \wedge \mathit{neg}(r)$.

Given two logic programs Π_1 and Π_2, we say that Π_1 *is independent of* Π_2, denoted $\Pi_2 \rhd \Pi_1$, if for each rule r_2 in Π_2, for each predicate symbol p occurring in $H(r_2)$, there is no rule r_1 in Π_1 such that p occurs in $B(r_1)$.

A term, or an atom, or a rule, or a program is said to be *ground* if no variable occurs in it. A ground instance of a term, or an atom, or a rule, or a program is obtained by replacing every variable occurrence by a ground term constructed by using function symbols appearing in Π. The set of all the ground instances of the rules of a program Π is denoted by $\mathit{ground}(\Pi)$. Note that if a program Π has function symbols with positive arity, then $\mathit{ground}(\Pi)$ may be infinite. However, as indicated at the beginning of Section 4, for our purposes we only need a finite subset of that infinite set.

An *interpretation* I of a program Π is a (finite or infinite) set of ground atoms. By $\overleftarrow{I}$ we denote the set $\{p \leftarrow \mid p \in I\}$ of facts. The *Gelfond-Lifschitz transformation* of $\mathit{ground}(\Pi)$ with respect to an interpretation I is the program $\mathit{ground}(\Pi)^I = \{\mathit{head}(r) \leftarrow \mathit{pos}(r) \mid r \in \mathit{ground}(\Pi) \text{ and } B^-(r) \cap I = \emptyset\}$. For any rule $r \in \mathit{ground}(\Pi)$, we say that I *satisfies* r if $(B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset)$ implies $H(r) \cap I \neq \emptyset$. An interpretation I is said to be an *answer set* of Π if I is a *minimal model* of $\mathit{ground}(\Pi)^I$, that is, I is a minimal set (with respect to set inclusion) which satisfies all rules in $\mathit{ground}(\Pi)^I$. The answer set semantics assigns to every program Π the set $\mathit{ans}(\Pi)$ of its answer sets.

Given a program $\Pi = \Pi_1 \cup \Pi_2$, the following fact holds [54]: if $\Pi_2 \rhd \Pi_1$, then $\mathit{ans}(\Pi) = \bigcup_{M \in \mathit{ans}(\Pi_1)} \mathit{ans}(\overleftarrow{M} \cup \Pi_2)$.

CHAPTER 2

Specifying Concurrent Programs

A *concurrent program* consists of a finite set of *processes* that are executed in parallel in a shared-memory environment, that is, processes that interact with each other through a *shared variable*. We assume that the shared variable ranges over a finite domain. With every process we associate a distinct *local variable* ranging over a finite domain which is the same for all processes. Every process may test and modify the shared variable and its own local variable by executing guarded commands.

Definition 10 (k-process concurrent program). For $k > 1$, let $\mathtt{x}_1$, …, $\mathtt{x}_k$ be *local variables* ranging over a finite domain $\mathcal{L}$ and let $\mathtt{y}$ be a *shared variable* ranging over a finite domain $\mathcal{D}$. For $i = 1, \ldots, k$, a *process* P_i is a guarded command of the form

$$P_i: \qquad \textit{true} \rightarrow \mathtt{if}\ gc_1\ [\!]\ \ldots\ [\!]\ gc_{n_i}\ \mathtt{fi}$$

where every guarded command gc in $gc_1\ [\!]\ \ldots\ [\!]\ gc_{n_i}$ is of the form:

$$gc: \qquad \mathtt{x}_i = l \wedge \mathtt{y} = d \quad \rightarrow \quad \mathtt{x}_i := l';\ \mathtt{y} := d'$$

with $\langle l, d\rangle \neq \langle l', d'\rangle$. We assume that, for $i = 1, \ldots, k$, the guards (that is, the expressions to the left of $\rightarrow$) of any two guarded commands of process P_i are mutually exclusive, that is, for all pairs $\langle l, d\rangle$, there is at most one occurrence of the guard '$\mathtt{x}_i = l \wedge \mathtt{y} = d$' in process P_i.
A *k-process concurrent program* C is a command of the form:

$$C: \qquad \mathtt{x}_1 := \bar{l}_1; \ldots; \mathtt{x}_k := \bar{l}_k;\ \mathtt{y} := \bar{d};\ \mathtt{do}\ P_1\ [\!]\ \ldots\ [\!]\ P_k\ \mathtt{od}$$

The $(k+1)$-tuple $\langle \bar{l}_1, \ldots, \bar{l}_k, \bar{d}\rangle$ is said to be the *initialization* of C. □

Example 11. Let $\mathcal{L}$ be $\{\mathtt{t},\mathtt{u}\}$ and $\mathcal{D}$ be $\{\mathtt{0},\mathtt{1}\}$. A 2-process concurrent program C is:

$\mathtt{x}_1 := \mathtt{t};\quad \mathtt{x}_2 := \mathtt{t};\quad \mathtt{y} := \mathtt{0};\quad \mathtt{do}\ P_1 \,[\!]\, P_2\ \mathtt{od}$

where P_1 and P_2 are defined as follows:

```
P1 : true → if                          P2 : true → if
        x1=t ∧ y=0 → x1:=u; y:=0             x2=t ∧ y=1 → x2:=u; y:=1
     [] x1=u ∧ y=0 → x1:=t; y:=1          [] x2=u ∧ y=1 → x2:=t; y:=0
     fi                                   fi
```

This program realizes a protocol which ensures mutual exclusion between the two processes P_1 and P_2. For $i = 1, 2$, process P_i either 'uses a resource' in its critical section, that is, the value of $\mathtt{x}_i$ is $\mathtt{u}$, or 'thinks' in its noncritical section, that is, the value of $\mathtt{x}_i$ is $\mathtt{t}$. The shared variable $\mathtt{y}$ gives the processes P_1 and P_2 the turn to enter the critical section: if $\mathtt{y}=\mathtt{0}$, process P_1 enters the critical section ($\mathtt{x}_1=\mathtt{u}$), while if $\mathtt{y}=\mathtt{1}$, process P_2 enters the critical section ($\mathtt{x}_2=\mathtt{u}$).

Note that in a real concurrent program, while P_i is in its noncritical (or critical) section it may execute arbitrary commands not affecting the values of the local and the shared variables. However, for the sake of simplicity, we omit such arbitrary commands and we will consider only those commands which are relevant to the interaction between processes. (A similar approach is taken in [25] where *synchronization skeletons* are considered.) □

Now we introduce the semantics of k-process concurrent programs by using Kripke structures. Given a k-process concurrent program C, a *state* of C is any $(k+1)$-tuple $\langle l_1, \ldots, l_k, d\rangle$, where: (i) the first k components are values for the local variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ of C, one local variable for each process P_i, and (ii) d is a value for the shared variable $\mathtt{y}$ of C. Given any state s, by $s(\mathtt{x}_i)$ we denote the value of the local variable of process P_i in state s and, similarly, by $s(\mathtt{y})$ we denote the value of the shared variable in state s.

Definition 11 (Reachability). Let C be a k-process concurrent program. We say that state s_2 is *one-step reachable* from state s_1, and we write $\mathit{Reach}(s_1, s_2)$, if there exists a process P_i, for some $i \in \{1, \ldots, k\}$, with a guarded command of the form: $\mathtt{x}_i = s_1(\mathtt{x}_i) \wedge \mathtt{y} = s_1(\mathtt{y}) \rightarrow \mathtt{x}_i := s_2(\mathtt{x}_i);\ \mathtt{y} := s_2(\mathtt{y})$, and for all $j \in \{1, \ldots, k\}$ different from i, $s_1(\mathtt{x}_j) = s_2(\mathtt{x}_j)$. We say that s_2 is *reachable* from s_1 if $\mathit{Reach}^*(s_1, s_2)$, where by Reach^* we denote the reflexive, transitive closure of *Reach*. □

Note that our definition of the transition relation *Reach* formalizes the *interleaving* semantics of guarded commands.

Definition 12 (Kripke structure associated with a k-process concurrent program). Let C be a k-process concurrent program of the form

$$C: \quad \mathtt{x}_1 := \bar{l}_1; \ldots ; \mathtt{x}_k := \bar{l}_k;\ \mathtt{y} := \bar{d};\ \mathtt{do}\ P_1\ [\!]\ \ldots\ [\!]\ P_k\ \mathtt{od}$$

Let *Reach* be the reachability relation associated with C which we assume to be total. The *Kripke structure* $\mathcal{K}$ *associated with* C is the 4-tuple $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$, where:

(i) $\mathcal{S} = \{s \mid \mathit{Reach}^*(s_0, s)\} \subseteq \mathcal{L}^k \times \mathcal{D}$ is the set of *reachable* states from s_0,

(ii) $\mathcal{S}_0 = \{s_0\} = \{\langle \bar{l}_1, \ldots, \bar{l}_k, \bar{d} \rangle\}$,

(iii) $\mathcal{R} = \mathit{Reach} \subseteq \mathcal{S} \times \mathcal{S}$, and

(iv) for all $\langle l_1, \ldots, l_k, d \rangle \in \mathcal{S}$, $\lambda(\langle l_1, \ldots, l_k, d \rangle) = \{\mathit{local}(P_1, l_1), \ldots, \mathit{local}(P_k, l_k), \mathit{shared}(d)\}$, where for $i = 1, \ldots, k$, the elementary proposition $\mathit{local}(P_i, l_i)$ denotes that the local variable $\mathtt{x}_i$ of process P_i has value l_i, and analogously, the elementary proposition $\mathit{shared}(d)$ denotes that the shared variable $\mathtt{y}$ has value d.

The set *Elem* of the elementary propositions is $\{\mathit{local}(P_i, l_i) \mid i = 1, \ldots, k\} \cup \{\mathit{shared}(d) \mid d \in \mathcal{D}\}$. □

Note that, since every state has a successor state, every concurrent program is a *nonterminating* program.

For every given state s, for every $i \in \{1, \ldots, k\}$, if $(\mathtt{x}_i = l \wedge \mathtt{y} = d \ \ \rightarrow \mathtt{x}_i := l';\ \mathtt{y} := d')$ is a guarded command in P_i such that $l = s(\mathtt{x}_i)$ and $d = s(\mathtt{y})$, then we say that P_i is *enabled* in s and the guard $\mathtt{x}_i = l \wedge \mathtt{y} = d$ *holds* in s.

Example 12. Given the 2-process concurrent program C of Example 11, the associated Kripke structure is depicted in Figure 15. We depict it as a graph whose nodes are the states reachable from the initial state $s_0 = \langle \mathtt{t}, \mathtt{t}, \mathtt{0} \rangle$. Each transition from state s to state t is associated with the guarded command whose guard holds in s. For the initial state s_0, we have that $\lambda(s_0) = \{\mathit{local}(P_1, \mathtt{t}), \mathit{local}(P_2, \mathtt{t}), \mathit{shared}(\mathtt{0})\}$ and, similarly, for the values of λ for the other states. □

Definition 13 (Satisfaction relation for a k-process concurrent program). Let C be a k-process concurrent program with initialization s_0, $\mathcal{K}$ be the Kripke structure associated with C, and φ be a CTL formula. We say that C *satisfies* φ, denoted $C \vDash \varphi$, if $\mathcal{K}, s_0 \vDash \varphi$. □

Example 13. Let us consider the 2-process concurrent program C defined in Example 11. The fact that the critical section associated with the value $\mathtt{u}$ of the local variable is executed in a mutually exclusive way, is formalized by the CTL formula $\varphi =_{def} \mathsf{AG}\,\neg(\mathit{local}(P_1, \mathtt{u}) \wedge \mathit{local}(P_2, \mathtt{u}))$. We have that $C \models \varphi$ holds

$$\begin{tikzcd}
 & & \langle \mathtt{u},\mathtt{t},\mathtt{0}\rangle \arrow[dr, "\mathtt{x}_1\!=\!\mathtt{u} \wedge \mathtt{y}\!=\!\mathtt{0} \rightarrow \mathtt{x}_1\!:=\!\mathtt{t};\mathtt{y}\!:=\!\mathtt{1}"] & \\
 {} \arrow[r] & \langle \mathtt{t},\mathtt{t},\mathtt{0}\rangle \arrow[ur, "\mathtt{x}_1\!=\!\mathtt{t} \wedge \mathtt{y}\!=\!\mathtt{0} \rightarrow \mathtt{x}_1\!:=\!\mathtt{u};\mathtt{y}\!:=\!\mathtt{0}"] & & \langle \mathtt{t},\mathtt{t},\mathtt{1}\rangle \arrow[dl, "\mathtt{x}_2\!=\!\mathtt{t} \wedge \mathtt{y}\!=\!\mathtt{1} \rightarrow \mathtt{x}_2\!:=\!\mathtt{u};\mathtt{y}\!:=\!\mathtt{1}"] \\
 & & \langle \mathtt{t},\mathtt{u},\mathtt{1}\rangle \arrow[ul, "\mathtt{x}_2\!=\!\mathtt{u} \wedge \mathtt{y}\!=\!\mathtt{1} \rightarrow \mathtt{x}_2\!:=\!\mathtt{t};\mathtt{y}\!:=\!\mathtt{0}"] &
\end{tikzcd}$$

Figure 15: The graph representing the transition relation *Reach* of the Kripke structure associated with the concurrent program of Example 11. Each arc is labelled by the guarded command which causes that transition according to Definition 11. The initial state is $s_0 = \langle \mathtt{t}, \mathtt{t}, \mathtt{0} \rangle$.

because for the Kripke structure $\mathcal{K}$ of Example 12 (see Figure 15), we have that $\mathcal{K}, s_0 \models \varphi$. Indeed, there is no path starting from the initial state $\langle \mathtt{t}, \mathtt{t}, \mathtt{0} \rangle$ which leads to either the state $\langle \mathtt{u}, \mathtt{u}, \mathtt{0} \rangle$ or the state $\langle \mathtt{u}, \mathtt{u}, \mathtt{1} \rangle$. □

In the literature (see, for instance, [7, 27, 56]) it is often considered the case where concurrent programs consist of similar processes, the similarity being determined by the fact that all processes follow the same cycling pattern of possible actions.

We formalize some *structural* properties which extend the notion of similarity. In particular, for any two distinct processes P_i and P_j in a concurrent program, we assume that process P_j can be obtained from process P_i by permuting the values of the shared variable `y`. For instance, in Example 11 the guarded commands in P_2 can be obtained from those in P_1 by interchanging `0` and `1`. Moreover, it is often the case that all processes of a given concurrent program C also share additional structural properties, such as the fact that the tests and the assignments performed on the local variables are the same for all processes in C. For instance, in Example 11 we have that both processes P_1 and P_2 may change state by changing the value of their local variables from `t` to `u` or from `u` to `t`.

Now we formalize those structural properties by introducing the *k-symmetric program structures.*

Definition 14 (*k*-symmetric program structure). For $k > 1$, let $\mathcal{L}$ be a finite domain for the local variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$, and $\mathcal{D}$ be a finite domain for the shared variable $\mathtt{y}$. A *k-symmetric program structure* $\sigma = \langle f, T, l_0, d_0 \rangle$ *over* $\mathcal{L}$ *and* $\mathcal{D}$ consists of: (i) a *k-generating function* $f \in \mathit{Perm}(\mathcal{D})$, which is either the identity function id or a generator of a cyclic group $\{\mathit{id}, f, f^2, \ldots, f^{k-1}\}$ of order k, (ii) a *local transition relation* $T \subseteq \mathcal{L} \times \mathcal{L}$ which is total over $\mathcal{L}$, (iii) an element $l_0 \in \mathcal{L}$, and (iv) an element $d_0 \in \mathcal{D}$. □

Definition 15 (*k*-process symmetric concurrent program). For any $k > 1$, let $\sigma = \langle f, T, l_0, d_0 \rangle$ be a k-symmetric program structure. A k-process concurrent program is said to be *symmetric* w.r.t. σ if it is of the form $\mathtt{x}_1 := l_0; \ldots; \mathtt{x}_k := l_0; \mathtt{y} := d_0;$ `do` $P_1 [\!] \ldots [\!] P_k$ `od` and, for all $i \in \{1, \ldots, k\}$, for all guarded commands gc of the form $\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'$, we have that:
(i) $\langle l, l' \rangle \in T$ and
(ii) gc is in P_i iff $(\mathtt{x}_{(i \bmod k)+1} = l \wedge \mathtt{y} = f(d) \rightarrow \mathtt{x}_{(i \bmod k)+1} := l'; \mathtt{y} := f(d'))$ is in $P_{(i \bmod k)+1}$. □

Example 14. Let us consider the 2-process concurrent program C of Example 11. The group $\mathit{Perm}(\mathcal{D})$ of permutations over $\mathcal{D} = \{\mathtt{0}, \mathtt{1}\}$ consists of the following two permutations: $\mathit{id} = \{\langle \mathtt{0}, \mathtt{0} \rangle, \langle \mathtt{1}, \mathtt{1} \rangle\}$ (that is, the identity permutation) and $f = \{\langle \mathtt{0}, \mathtt{1} \rangle, \langle \mathtt{1}, \mathtt{0} \rangle\}$. The program C is symmetric w.r.t. the 2-symmetric program structure $\langle f, T, \mathtt{t}, \mathtt{0} \rangle$, where the local transition relation T is $\{\langle \mathtt{t}, \mathtt{u} \rangle, \langle \mathtt{u}, \mathtt{t} \rangle\}$. Indeed, its initialization is: $\mathtt{x}_1 := \mathtt{t}$; $\mathtt{x}_2 := \mathtt{t}$; $\mathtt{y} := \mathtt{0}$, and processes P_1 and P_2 are as follows:

```
P_1 : true → if
        x_1 = t ∧ y = 0 → x_1 := u; y := 0
    [] x_1 = u ∧ y = 0 → x_1 := t; y := 1
fi

P_2 : true → if
        x_2 = t ∧ y = f(0) → x_2 := u; y := f(0)
    [] x_2 = u ∧ y = f(0) → x_2 := t; y := f(1)
 fi
```
□

CHAPTER 3

Synthesizing Concurrent Programs

Now we present our method based on Answer Set Programming for synthesizing a k-process symmetric concurrent program from a CTL formula encoding a given behavioural property and a k-symmetric program structure encoding a given structural property.

Definition 16 (The synthesis problem). Given a CTL formula φ and a k-symmetric program structure σ over the finite domains $\mathcal{L}$ and $\mathcal{D}$, the *synthesis problem* consists in finding a k-process concurrent program C such that $C \vDash \varphi$ and C is symmetric with respect to σ. $\Box$

The synthesis problem can be solved by applying the following two-step procedure:

(*Step* 1) we generate a k-process symmetric concurrent program C, and

(*Step* 2) we verify whether or not C satisfies a given behavioural property φ.

By Definition 15, from any process P_i, with $i = 1, \ldots, k$, we derive process $P_{(i \bmod k)+1}$ by applying the k-generating function f to the guarded commands of P_i, thereby deriving the guarded commands of $P_{(i \bmod k)+1}$. Thus, Step 1 can be performed by generating process P_1 and using f for generating the other $k{-}1$ processes. Then Step 2 reduces to the test of the satisfiability relation $\mathcal{K}, s_0 \vDash \varphi$, where: (i) $\mathcal{K}$ is the Kripke structure associated with C, and (ii) state s_0 is the initial state of $\mathcal{K}$ corresponding to the initialization of C.

We present a solution to the synthesis problem in a purely declarative manner by reducing it to the problem of computing the answer sets of a logic program Π encoding an instance of the synthesis problem. The logic program Π is the union of a program Π_σ which encodes a structural property σ and a program Π_φ which encodes a behavioural property φ.

In Theorem 9 we will prove that every answer set of Π encodes a k-process concurrent program satisfying φ and which is symmetric w.r.t. σ. We have that Π_σ is independent of Π_φ (that is, $\Pi_\varphi \rhd \Pi_\sigma$) and, thus, we can first compute the answer sets of Π_σ and then use those answer sets, together with program Π_φ, to test whether or not the encoded k-symmetric concurrent program satisfies φ.

Programs Π_σ and Π_φ are introduced by the following Definitions 17 and 18, respectively.

Definition 17 (Logic program encoding a structural property). Let $\sigma = \langle f, T, l_0, d_0\rangle$ be a k-symmetric program structure over the finite domains $\mathcal{L}$ and $\mathcal{D}$ and s_0 be the $(k+1)$-tuple $\langle l_0, \ldots, l_0, d_0\rangle$. The logic program Π_σ is as follows:

1.1 $\mathit{enabled}(1, X_1, Y) \vee \mathit{disabled}(1, X_1, Y) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y\rangle)$
1.2 $\mathit{enabled}(2, X, Y) \leftarrow \mathit{gc}(2, X, Y, X', Y')$

$\vdots$

1.k $\mathit{enabled}(k, X, Y) \leftarrow \mathit{gc}(k, X, Y, X', Y')$
2.1 $\mathit{gc}(1, X, Y, X_1, Y_1) \vee \ldots \vee \mathit{gc}(1, X, Y, X_m, Y_m) \leftarrow \mathit{enabled}(1, X, Y) \wedge \mathit{candidates}(X, Y, [\langle X_1, Y_1\rangle, \ldots, \langle X_m, Y_m\rangle])$
2.2 $\mathit{gc}(2, X, Z, X', Z') \leftarrow \mathit{gc}(1, X, Y, X', Y') \wedge \mathit{perm}(Y, Z) \wedge \mathit{perm}(Y', Z')$

$\vdots$

2.k $\mathit{gc}(k, X, Z, X', Z') \leftarrow \mathit{gc}(k-1, X, Y, X', Y') \wedge \mathit{perm}(Y, Z) \wedge \mathit{perm}(Y', Z')$
3.1 $\mathit{reachable}(s_0) \leftarrow$
3.2 $\mathit{reachable}(\langle X_1, \ldots, X_k, Y\rangle) \leftarrow \mathit{tr}(\langle X'_1, \ldots, X'_k, Y'\rangle, \langle X_1, \ldots, X_k, Y\rangle)$
4.1 $\mathit{tr}(\langle X_1, \ldots, X_k, Y\rangle, \langle X'_1, \ldots, X_k, Y'\rangle) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y\rangle) \wedge \mathit{gc}(1, X_1, Y, X'_1, Y')$

$\vdots$

4.k $\mathit{tr}(\langle X_1, \ldots, X_k, Y\rangle, \langle X_1, \ldots, X'_k, Y'\rangle) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y\rangle) \wedge \mathit{gc}(k, X_k, Y, X'_k, Y')$
5. $\leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y\rangle) \wedge \mathtt{not}\ \mathit{enabled}(1, X_1, Y) \wedge \ldots \wedge \mathtt{not}\ \mathit{enabled}(k, X_k, Y)$

together with the following two sets of ground facts:

(i) $\{\mathit{candidates}(l, d, L(l, d)) \leftarrow \mid l \in \mathcal{L} \wedge d \in \mathcal{D}\}$, where $L(l, d)$ is any list representing the set of pairs $\{\langle l', d'\rangle \mid \langle l, l'\rangle \in T \wedge d' \in \mathcal{D} \wedge \langle l, d\rangle \neq \langle l', d'\rangle\}$

(ii) $\{\mathit{perm}(d, d') \leftarrow \mid d, d' \in \mathcal{D} \wedge f(d) = d'\}$. □

In this program, for $i=1,\ldots,k$, the predicate $gc(i,l,d,l',d')$ holds iff in process P_i there exists the guarded command $\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'$ (see also Definition 19).

Rule 1.1 states that in every reachable state, process P_1 is either enabled (that is, one of its guards holds) or disabled. Rule 1.1 is used to derive atoms either of the form $\mathit{enabled}(1,X_1,Y)$ or of the form $\mathit{disabled}(1,X_1,Y)$. If an atom of the form $\mathit{enabled}(1,X_1,Y)$ is derived, then a guarded command for process P_1 (that is, an atom of the form $gc(1,X,Y,X_i,Y_i)$) is generated by using Rule 2.1. Note that, without Rule 1.1, no atom for the predicates *enabled* and *gc* could be generated and, therefore, no concurrent program would be synthesized.

Rules 1.i, with $i = 2,\ldots,k$, state that any process P_i is enabled in state s if P_i has a guarded command of the form $\mathtt{x}_i = X \wedge \mathtt{y} = Y \rightarrow \mathtt{x}_i := X'$; $\mathtt{y} := Y'$, for some values of X' and Y', such that $X = s(\mathtt{x}_i)$ and $Y = s(\mathtt{y})$.

The disjunctive Rule 2.1 generates a guarded command for process P_1 by first enumerating all candidate guarded commands for that process (through the predicate *candidates*) and then selecting one candidate which corresponds to a disjunct of its head. Each guarded command consists of the guard $\mathtt{x}_1 = X \wedge \mathtt{y} = Y$, encoded by using the atom $\mathit{enabled}(1,X,Y)$, and a command $\mathtt{x}_1 := X_i$; $\mathtt{y} := Y_i$, encoded by a pair $\langle X_i, Y_i\rangle$ in the list which is the third argument of $\mathit{candidates}(X,Y,L(l,d))$.

The number m of pairs $\langle X_i, Y_i\rangle$ in the list $L(l,d)$ is uniquely determined by the values l and d of the variables X and Y, respectively, in $\mathit{enabled}(1,X,Y)$. (It can be shown that $|\mathcal{D}|-1 \leq m \leq |\mathcal{L}|\cdot|\mathcal{D}|-1$.) Thus, Rule 2.1 actually stands for a set of rules, one rule for each value of m, and this set of rules can effectively be derived only when the set of facts for the predicate *candidates* is computed.

For instance, let us consider the sets $T = \{\langle \mathtt{a},\mathtt{b}\rangle, \langle \mathtt{a},\mathtt{a}\rangle, \langle \mathtt{b},\mathtt{a}\rangle\}$ and $\mathcal{D} = \{\mathtt{0},\mathtt{1}\}$. For $X = \mathtt{b}$, $Y = \mathtt{0}$, we have that $\mathit{candidates}(\mathtt{b},\mathtt{0},[\langle \mathtt{a},\mathtt{0}\rangle,\langle \mathtt{a},\mathtt{1}\rangle])$ holds (recall that a guarded command should change either the value of the local variable or the value of the shared variable), and for $X = \mathtt{a}$, $Y = \mathtt{0}$, we have that $\mathit{candidates}(\mathtt{a},\mathtt{0},[\langle \mathtt{a},\mathtt{1}\rangle,\langle \mathtt{b},\mathtt{0}\rangle,\langle \mathtt{b},\mathtt{1}\rangle])$ holds. Hence, when $Y = \mathtt{0}$, we have two instances of Rule 2.1, one for $m=2$ and one for $m=3$.

Rules 2.2–2.k realize Definition 15. In particular, they allow us to derive the guarded command for processes $P_2,\ldots,P_k$ from the guarded commands generated for process P_1. Note that, due to our definition of a symmetric program structure, the subscript of the process used for the initial choice (1 in our case) is immaterial, in the sense that any other choice for that subscript produces a solution satisfying the same behavioural and structural properties.

Rules 3.1, 3.2, and 4.1–4.k define, in a mutually recursive way, the reachability relation (encoded by the predicate *reachable*) and the transition relation $\mathcal{R}$ (encoded by the predicate *tr*) of the Kripke structure associated with the concurrent program to be synthesized.

Rule 5 is an integrity constraint enforcing that any answer set of Π_σ is a model of $\Pi_\sigma - \{\text{Rule 5}\}$ which does not satisfy the body of Rule 5. Thus, Rule 5 guarantees that the transition relation $\mathcal{R}$ is total, that is, in every reachable state there exists at least one enabled process.

Now let us present the logic program Π_φ which encodes a given behavioural property φ. Note that program Π_φ depends on program Π_σ for the definition of the transition relation $tr(S,T)$ and for the initial state s_0, which is assumed to be the $(k+1)$-tuple $\langle l_0, \ldots, l_0, d_0 \rangle$.

Definition 18 (Logic program encoding a behavioural property). Let φ be a CTL formula. The logic program Π_φ encoding φ is as follows:

1. $\leftarrow \texttt{not}\ sat(s_0, \varphi)$
2. $sat(S,F) \leftarrow elem(S,F)$
3. $sat(S, not(F)) \leftarrow \texttt{not}\ sat(S,F)$
4. $sat(S, and(F_1,F_2)) \leftarrow sat(S,F_1) \wedge sat(S,F_2)$
5. $sat(S, ex(F)) \leftarrow tr(S,T) \wedge sat(T,F)$
6. $sat(S, eu(F_1,F_2)) \leftarrow sat(S,F_2)$
7. $sat(S, eu(F_1,F_2)) \leftarrow sat(S,F_1) \wedge tr(S,T) \wedge sat(T, eu(F_1,F_2))$
8. $sat(S, eg(F)) \leftarrow satpath(S,T,F) \wedge satpath(T,T,F)$
9. $satpath(S,T,F) \leftarrow sat(S,F) \wedge tr(S,T)$
10. $satpath(S,V,F) \leftarrow sat(S,F) \wedge tr(S,T) \wedge satpath(T,V,F)$

together with the following two sets of ground facts:

(i) $\{elem(s, local(P_i, l)) \leftarrow \mid 1 \le i \le k \wedge s \in \mathcal{L}^k \times \mathcal{D} \wedge s(\mathtt{x}_i) = l\}$

(ii) $\{elem(s, shared(d)) \leftarrow \mid s \in \mathcal{L}^k \times \mathcal{D} \wedge s(\mathtt{y}) = d\}$. □

Note that in the ground facts defining *elem*, for $i = 1, \ldots, k$, by $s(\mathtt{x}_i)$ we denote the i-th component of s and by $s(\mathtt{y})$ we denote the $(k+1)$-th component of s (see Chapter 2 for this notational convention). In Rule 1 of program Π_φ, by abuse of language, we use φ to denote the ground term representing the CTL formula φ. In particular, in the ground term φ we use the function symbols *not*, *and*, *ex*, *eu* and *eg* to denote the operators $\neg$, $\wedge$, EX, EU, and EG, respectively.

Rules 2–10, taken from [117], encode the semantics of CTL formulas as follows: (*i*) $sat(s,\psi)$ holds iff the formula ψ holds in state s, and (*ii*) $satpath(s,t,\psi)$ holds iff there exists a path from state s to state t such that every state in that path (except possibly the last one) satisfies the formula ψ. Rule 1 is an integrity constraint enforcing that any answer set of Π is a model of $(\Pi_\varphi \cup \Pi_\sigma) - \{\text{Rule 1}\}$ satisfying $sat(s_0, \varphi)$.

Now we establish the correctness (that is, the soundness and completeness) of our synthesis procedure. It relates the k-process symmetric (with respect to σ) concurrent programs satisfying φ with the answer sets of the logic program $\Pi_\varphi \cup \Pi_\sigma$. Let us first introduce the following definition.

Definition 19 (Encoding of a k-process concurrent program). Let C be a k-process concurrent program of the form $\mathtt{x}_1 := \overline{l}_1; \ldots; \mathtt{x}_k := \overline{l}_k;\ \mathtt{y} := \overline{d};$ `do` $P_1\ [\!]\ \ldots\ [\!]\ P_k$ `od`. Let M be a set of ground atoms. We say that M *encodes* C if, for all i, l, d, l', d', the following holds:

$gc(i, l, d, l', d') \in M$ iff $(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d')$ is a guarded command in P_i. □

Theorem 9 (Soundness and completeness of synthesis). Let φ be a CTL formula and σ be a k-symmetric program structure over the finite domains $\mathcal{L}$ and $\mathcal{D}$. Then, there exists a k-process concurrent program C such that (i) $C \vDash \varphi$ and (ii) C is symmetric w.r.t. σ iff there exists an answer set $M \in ans(\Pi_\varphi \cup \Pi_\sigma)$ such that M encodes C. □

The following theorem establishes the complexity of our synthesis procedure as a function of the synthesis parameters, that is, (i) the number k of processes, (ii) the size $|\varphi|$ of the CTL behavioural property φ defined to be the number of operators and elementary propositions occurring in φ, and (iii) the cardinalities of $\mathcal{L}$ and $\mathcal{D}$ which are the domains of f and T, respectively. When we state the complexity result with respect to one parameter, we assume that the others remain constant.

Theorem 10 (Complexity of synthesis). For any number $k > 1$ of processes, for any symmetric program structure σ over $\mathcal{L}$ and $\mathcal{D}$, and for any CTL formula φ, an answer set of the logic program $\Pi_\varphi \cup \Pi_\sigma$ can be computed in (i) exponential time w.r.t. k, (ii) linear time w.r.t. $|\varphi|$, and (iii) nondeterministic polynomial time w.r.t. $|\mathcal{L}|$ and w.r.t. $|\mathcal{D}|$. □

It is known (see, for instance, [96]) that the problem of synthesis from a CTL specification φ is EXPTIME-complete w.r.t. $|\varphi|$. In order to compare the complexity of our synthesis procedure with that of other techniques which can be found in the literature [7, 8, 25, 96, 80], note that the parameters of our synthesis procedure are not mutually independent. In particular, as we will see in the following section, the usual behavioural properties considered for the mutual exclusion problem, determine a CTL specification whose size depends on the number k of processes. However, since our ASP synthesis procedure has time complexity which is exponential w.r.t. k, it turns out that our translation yields a synthesis procedure which still belongs to the EXPTIME class and, thus, it matches the complexity of the synthesis problem.

CHAPTER 4

Synthesis examples

In this chapter we present some experimental results obtained by applying our procedure for the synthesis of various mutual exclusion protocols.

In order to compute the answer sets of a logic program P with an ASP solver, we should first construct the set $ground(P)$. This set is constructed by a *grounder* which is *either* a standalone tool, such as *gringo* [53] or *lparse* [133], independent of the ASP solver, *or* is a built-in module of the ASP solver, as in the *DLV* system [99].

If a logic program P has function symbols with positive arity, then $ground(P)$ is infinite. Thus, in particular, $ground(\Pi)$ is infinite. However, in order to compute the answer sets of Π, we only need some finite subsets of $ground(\Pi)$. These subsets are constructed by most grounders by means of the so called *domain predicates*, which specify the finite domains over which the variables should range [53, 99, 133].

In our case, a finite set of ground rules is obtained from program Π_φ by introducing in the body of each of the Rules 2–10 a domain predicate so that terms representing CTL formulas are restricted to range over subterms of φ. (Here and in what follows, when we refer to a subterm, we mean a non necessarily proper subterm.) In particular, a rule of the form $sat(S, \psi) \leftarrow Body$ is replaced by $sat(S, \psi) \leftarrow Body \wedge d(\psi)$, where d is the domain predicate defined by the set $\{d(\psi)\leftarrow \mid \psi \text{ is a subterm of } \varphi\}$ of ground facts. The correctness of this replacement relies on the fact that, in order to prove $sat(s_0, \varphi)$ by using Rules 2–10, it is sufficient to consider only the instances of these rules where subterms of φ occur.

Note that, by using a grounder after the introduction of domain predicates, we get a set of ground instances of Rules 2–10 whose cardinality is linear in the number of subterms of φ and, hence, in the size of φ. This fact is relevant for the complexity results stated in Theorem 10.

In our synthesis experiments, in order to define the k-symmetric program structures of the programs to be synthesized, we have made the following choices for: (i) the domain $\mathcal{L}$ of the local variables $\mathtt{x}_i$'s, (ii) the domain $\mathcal{D}$ of the shared variable $\mathtt{y}$, (iii) the k-generating function f, (iv) the set T, (v) the value of $l_0 \in \mathcal{L}$, and (vi) the value of $d_0 \in \mathcal{D}$.

We have taken the domain $\mathcal{L}$ to be $\{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, where $\mathtt{t}$ represents the *noncritical section*, $\mathtt{w}$ represents the *waiting section*, and $\mathtt{u}$ represents the *critical section*.

We have taken the domain $\mathcal{D}$ to be $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{n}\}$, where $\mathtt{n}$ depends on: (i) the number k of the processes in the concurrent program to be synthesized, and (ii) the properties that the concurrent program should satisfy. At the beginning of every synthesis experiment we have taken $\mathtt{n} = \mathtt{1}$ and, if the synthesis failed, we have increased the value of $\mathtt{n}$ by one unity at a time, hoping for a successful synthesis with a larger value of $\mathtt{n}$.

We have taken the k-generating function f to be either (i) the identity function *id*, or (ii) a permutation among the $|\mathcal{D}|!/(k \cdot (|\mathcal{D}| - k)!)$ permutations of order k defined over $\mathcal{D}$.

We have taken the local transition relation T to be $\{\langle \mathtt{t}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{u}\rangle, \langle \mathtt{u}, \mathtt{t}\rangle\}$. The pair $\langle \mathtt{t}, \mathtt{w}\rangle$ denotes that, once the noncritical section $\mathtt{t}$ has been executed, a process may enter the waiting section $\mathtt{w}$. The pairs $\langle \mathtt{w}, \mathtt{w}\rangle$ and $\langle \mathtt{w}, \mathtt{u}\rangle$ denote that a process may repeat (possibly an unbounded number of times) the execution of its waiting section $\mathtt{w}$ and then may enter its critical section $\mathtt{u}$. The pair $\langle \mathtt{u}, \mathtt{t}\rangle$ denotes that, once the critical section $\mathtt{u}$ has been executed, a process may enter its noncritical section $\mathtt{t}$.

Finally, we have taken l_0 to be $\mathtt{t}$ and d_0 to be $\mathtt{0}$.

For $k = 2, \ldots, 6$, we have synthesized (see Column 1 of Table 4.1) various k-process symmetric concurrent programs of the form $\mathtt{x}_1 := \mathtt{t};\ \ldots;\ \mathtt{x}_k := \mathtt{t};\ \mathtt{y} := \mathtt{0};\ \mathtt{do}\ P_1\ [\!]\ \ldots\ [\!]\ P_k\ \mathtt{od}$, which satisfies some behavioural properties among those defined by the following CTL formulas (see Column 2 of Table 4.1).

(i) *Mutual Exclusion*, that is, it is not the case that process P_i is in its critical section ($\mathtt{x}_i = \mathtt{u}$), and process P_j is in its critical section ($\mathtt{x}_j = \mathtt{u}$) at the same time: for all i, j in $\{1, \ldots, k\}$, with $i \neq j$,

$$\mathsf{AG}\,\neg(\mathit{local}(\mathtt{P}_i, \mathtt{u}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u})) \qquad (\mathit{ME})$$

(ii) *Progression and Starvation Freedom*, that is, (progression) every process P_i which is in the noncritical section, may enter its waiting section (that is, modify the local variable $\mathtt{x}_i$ from $\mathtt{t}$ to $\mathtt{w}$), thereby requesting to enter the critical section, and (starvation freedom) if a process P_i is in waiting section ($\mathtt{x}_i = \mathtt{w}$), then after a finite amount of time, it will enter its critical section ($\mathtt{x}_i = \mathtt{u}$): for all i in $\{1, \ldots, k\}$,

$$\mathsf{AG}\,((\mathit{local}(\mathtt{P}_i, \mathtt{t}) \rightarrow \mathsf{EX}\,\mathit{local}(\mathtt{P}_i, \mathtt{w})) \wedge (\mathit{local}(\mathtt{P}_i, \mathtt{w}) \rightarrow \mathsf{AF}\,\mathit{local}(\mathtt{P}_i, \mathtt{u}))) \qquad (\mathit{SF})$$

(iii) *Bounded Overtaking*, that is, while process P_i is in its waiting section, every other process P_j leaves its critical section *at most once*, that is, P_j should not be in its critical section u and then in its waiting section w and then again in its critical section u, while P_i is always in its waiting section w (see the underlined subformulas): for all i, j in $\{1, \ldots, k\}$, with $i \neq j$,

$$\begin{aligned}\mathsf{AG}\,\neg\,[&\underline{\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u})} \wedge \\ &\mathsf{E}\,[\mathit{local}(\mathtt{P}_i, \mathtt{w})\ \mathsf{U}\ (\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{w}) \wedge \\ &\qquad \mathsf{E}\,[\mathit{local}(\mathtt{P}_i, \mathtt{w})\ \mathsf{U}\ \underline{(\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u}))}])]]\end{aligned} \qquad (\mathit{BO})$$

(iv) *Maximal Reactivity*, that is, if process P_i is in its waiting section and all other processes are in their noncritical sections, then in the next state P_i will be in its critical section: for all i in $\{1, \ldots, k\}$,

$$\mathsf{AG}\,((\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \textstyle\bigwedge_{j \in \{1,\ldots,k\}-\{i\}} \mathit{local}(\mathtt{P}_j, \mathtt{t})) \rightarrow \mathsf{EX}\,\mathit{local}(\mathtt{P}_i, \mathtt{u})) \qquad (\mathit{MR})$$

First, we have synthesized a simple protocol, called 2-*mutex*-1, for two processes enjoying the mutual exclusion property (see row 1 of Table 4.1), and then we synthesized various other protocols for two or more processes which enjoy other properties. In that table the identifier *k-mutex-p* occurring in the first column, denotes the synthesized protocol for k processes satisfying the $p\,(\geq 1)$ behavioural properties listed in the second column Properties. For instance, program 2-*mutex*-4 is the synthesized protocol for 2 processes which enjoys the four behavioural properties *ME*, *SF*, *BO*, and *MR*. In each row of Table 4.1 we have shown the minimal cardinality (in Column $|\mathcal{D}|$) and the k-generating function (in Column f) for which the synthesis of the program of that row succeeds.

The synthesis of program 2-*mutex*-1 succeeds with $|\mathcal{D}|$=2 and both the identity function and the permutation $f_1 = \{\langle 0, 1\rangle, \langle 1, 0\rangle\}$ (see rows 1 and 2). The syntheses of programs 2-*mutex*-2 and 2-*mutex*-3 fail for $|\mathcal{D}| = 2$ and the identity function, but they succeed for $|\mathcal{D}| = 2$ and f_1 (see rows 3 and 4). The synthesis of 2-*mutex*-4 fails for $|\mathcal{D}| = 2$ and any choice of a 2-generating function. Thus, we increased $|\mathcal{D}|$ from 2 to 3. For $|\mathcal{D}| = 3$ and the identity function the synthesis fails, but it succeeds for the permutation $f_2 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 2, 2\rangle\}$ of order 2 (see row 5). If we use different permutations of order 2, instead of f_2, we get programs which are equal to the program 2-*mutex*-4 (presented in Figure 17), modulo a permutation of the values of the shared variable y.

The synthesis of 3-*mutex*-1 succeeds for $|\mathcal{D}|=2$ and the identity function (see row 6). The synthesis of 3-*mutex*-2 fails for $|\mathcal{D}| = 2$ (the only choice for the 3-generating function is the identity function) and, thus, we increased $|\mathcal{D}|$ from 2 to 3. By using $|\mathcal{D}| = 3$ and the identity function, the synthesis fails, but it succeeds for $|\mathcal{D}|=3$ and the permutation $f_3=\{\langle 0,1\rangle,\langle 1,2\rangle,\langle 2,0\rangle\}$ of order 3 (see row 7). This synthesis succeeds also by using different permutations of order 3, and in all these cases we get programs which are equal to 3-*mutex*-2, modulo a permutation of the values of the shared variable y.

The synthesis of 3-*mutex*-3 (see row 8) is analogous to that of 3-*mutex*-2 to which row 7 refers.

The synthesis of 3-*mutex*-4 fails for $|\mathcal{D}| = 4, 5$, and 6, while it succeeds for $|\mathcal{D}| = 7$ and the permutation $f_4 = \{\langle 0,1\rangle,\langle 1,2\rangle,\langle 2,0\rangle,\langle 3,4\rangle,\langle 4,5\rangle,\langle 5,3\rangle,\langle 6,6\rangle\}$ which is of order 3 (see row 9).

The last rows 10, 11, and 12 of Table 4.1 refer, respectively, to the programs 4-*mutex*-1, 5-*mutex*-1, and 6-*mutex*-1 whose syntheses succeed for $|\mathcal{D}| = 2$ and the identity function.

In Figure 17 we present the synthesized program, called 2-*mutex*-4, for the 2-process mutual exclusion problem described in Example 13. In Figure 18 we present the transition relation of the associated Kripke structure. Program 2-*mutex*-4 is basically the same as Peterson algorithm [120], but, instead of using three shared variables, each of which ranges over a domain of two values, program 2-*mutex*-4 uses two local variables $\mathtt{x}_1$ and $\mathtt{x}_2$ which range over $\{\mathtt{t},\mathtt{w},\mathtt{u}\}$, and a single shared variable y which ranges over $\{0,1,2\}$.

The comparison between Peterson algorithm and our program 2-*mutex*-4 is illustrated in Figure 16, where in the upper part we have presented the original Peterson algorithm for two processes and in the lower part our synthesized Peterson-like algorithm derived by hand from the transition relation of program 2-*mutex*-4 depicted in Figure 18. Note that in Peterson algorithm the three shared variables are assigned constant values, while in our algorithm we pay the price of using a single shared variable y by the need of performing some operations on that variable.

However, in Peterson algorithm if a process, say P_1, is in its waiting section and the other process P_2 fails after assigning to q and never does the assignment to s, then P_1 cannot enter its critical section. This problem can be avoided by assuming that the sequence of assignments to q and s is atomic. In our algorithm this problem does not arise simply because we have not a sequence of

Program	Properties	$\lvert\mathcal{D}\rvert$	f	$\lvert ans(\Pi)\rvert$	Time
(1) 2-*mutex*-1	*ME*	2	*id*	10	0.01
(2) 2-*mutex*-1	*ME*	2	f_1	10	0.01
(3) 2-*mutex*-2	*ME*, *SF*	2	f_1	2	0.03
(4) 2-*mutex*-3	*ME*, *SF*, *BO*	2	f_1	2	0.05
(5) 2-*mutex*-4	*ME*, *SF*, *BO*, *MR*	3	f_2	2	0.17
(6) 3-*mutex*-1	*ME*	2	*id*	9	0.05
(7) 3-*mutex*-2	*ME*, *SF*	3	f_3	6	3.49
(8) 3-*mutex*-3	*ME*, *SF*, *BO*	3	f_3	4	4.32
(9) 3-*mutex*-4	*ME*, *SF*, *BO*, *MR*	7	f_4	2916	$\approx$ 4.4 hours
(10) 4-*mutex*-1	*ME*	2	*id*	9	0.35
(11) 5-*mutex*-1	*ME*	2	*id*	9	2.89
(12) 6-*mutex*-1	*ME*	2	*id*	9	20.43

Table 4.1: Column 'Program' shows the names of the synthesized programs. *k-mutex-p* is the name of the k-process program satisfying the p behavioural properties shown in column 'Properties'. Column $|\mathcal{D}|$ shows the cardinality of the domain $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{n}\}$ of the shared variable $\mathtt{y}$. Column f shows the k-generating function used for the synthesis. Column $|ans(\Pi)|$ shows the number of answer sets of $\Pi = \Pi_\varphi \cup \Pi_\sigma$. Column 'Time' shows the time expressed in seconds (unless otherwise specified) to generate all answer sets of Π, by using the ASP solver *claspD* [53].

assignments, but a single assignment to $\mathtt{y}$.

Let us briefly explain our hand derivation of the Peterson-like algorithm from the algorithm described by guarded commands in Figure 17. Let us consider process P_1. We have that:

(i) when P_1 enters the waiting section or leaves the critical section, $\mathtt{y}$ is modified as follows: $\mathtt{y} := \textit{if } \mathtt{y}=\mathtt{2} \textit{ then } \mathtt{1} \textit{ else } \mathtt{2}$ (see the guarded commands (1), (2), (3), (6), and (7)), and

(ii) when P_1 enters the critical section, $\mathtt{y}$ should be different from $\mathtt{1}$ and the value of $\mathtt{y}$ is not modified (see the guarded commands (4) and (5)). For process P_2 we replace $\mathtt{1}$ by $\mathtt{0}$.

As indicated in Figure 16 the two assignments to $\mathtt{y}$ (that is, $\mathtt{y} := \textit{if } \mathtt{y}=\mathtt{2} \textit{ then } \mathtt{1} \textit{ else } \mathtt{2}$ for P_1, and $\mathtt{y} := \textit{if } \mathtt{y} = \mathtt{2} \textit{ then } \mathtt{0} \textit{ else } \mathtt{2}$ for P_2) can be expressed as assignments in Kleene 3-valued logic whose values are $\mathtt{0}$, $\mathtt{1}$, and $\mathtt{2}$. In that logic we have that:

(i) $\neg \mathtt{x} =_{def} 2 - \mathtt{x}$,
(ii) $\mathtt{x} \wedge \mathtt{y} =_{def} \min(\mathtt{x}, \mathtt{y})$,
(iii) $\mathtt{x} \vee \mathtt{y} =_{def} \max(\mathtt{x}, \mathtt{y})$, and
(iv) $\mathtt{x} \rightarrow \mathtt{y} =_{def}$ *if* $\mathtt{x} \leq \mathtt{y}$ *then* $\mathtt{2}$ *else* $\mathtt{0}$. Note that in that logic $\mathtt{x} \rightarrow \mathtt{y} \neq \neg\mathtt{x} \vee \mathtt{y}$.

In particular, for proess P_1 $\mathtt{y} :=$ *if* $\mathtt{y}=\mathtt{2}$ *then* $\mathtt{1}$ is equal to $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1}) \vee \mathtt{1}$ and for process P_2 is equal to $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1})$.

Peterson Algorithm for the 2 processes P_1 and P_2 [120]

```
              $q_1 := false$;   $q_2 := false$;   $s := 1$;
$P_1$ :  while $true$ do                  | $P_2$ :  while $true$ do
   $l_1$ :  non-critical section 1;       |   $m_1$ :  non-critical section 2;
   $l_2$ :  $q_1 := true$; $s := 1$;      |   $m_2$ :  $q_2 := true$; $s := 2$;
   $l_3$ :  await $(\neg q_2) \vee (s = 2)$; |   $m_3$ :  await $(\neg q_1) \vee (s = 1)$;
   $l_4$ :  critical section 1;           |   $m_4$ :  critical section 2;
   $l_5$ :  $q_1 := false$;  od           |   $m_5$ :  $q_2 := false$;  od
```

Synthesized Algorithm for the 2 processes P_1 and P_2 (2-*mutex*-4)

```
                        $\mathtt{y} := \mathtt{0}$;
$P_1$ :  while $true$ do                  | $P_2$ :  while $true$ do
   $l_1$ :  non-critical section 1;       |   $m_1$ :  non-critical section 2;
   $l_2$ :  $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1}) \vee \mathtt{1}$;   |   $m_2$ :  $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1})$;
   $l_3$ :  await $\mathtt{y} \neq \mathtt{1}$;  |   $m_3$ :  await $\mathtt{y} \neq \mathtt{0}$;
   $l_4$ :  critical section 1;           |   $m_4$ :  critical section 2;
   $l_5$ :  $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1}) \vee \mathtt{1}$;  od |   $m_4$ :  $\mathtt{y} := (\mathtt{y} \rightarrow \mathtt{1})$;  od
```

Figure 16: The original Peterson algorithm for 2 processes (above) compared with our synthesized Peterson-like algorithm for 2 processes 2-*mutex*-4 (below) derived by hand from the transition relation of Figure 18. Implication and disjunctions are performed in Kleene 3-valued logic.

4.1 Comparison of ASP Solvers on the Synthesis Examples

We have implemented our synthesis procedure by using the following ASP solvers:

```
          $P_1$ : $true \rightarrow$ if                                $P_2$ : $true \rightarrow$ if
(1)            x1=t ∧ y=0 → x1:=w; y:=2                  x2=t ∧ y=0 → x2:=w; y:=2
(2)         [] x1=t ∧ y=1 → x1:=w; y:=2               [] x2=t ∧ y=1 → x2:=w; y:=2
(3)         [] x1=t ∧ y=2 → x1:=w; y:=1               [] x2=t ∧ y=2 → x2:=w; y:=0
(4)         [] x1=w ∧ y=0 → x1:=u; y:=0               [] x2=w ∧ y=1 → x2:=u; y:=1
(5)         [] x1=w ∧ y=2 → x1:=u; y:=2               [] x2=w ∧ y=2 → x2:=u; y:=2
(6)         [] x1=u ∧ y=2 → x1:=t; y:=1               [] x2=u ∧ y=2 → x2:=t; y:=0
(7)         [] x1=u ∧ y=0 → x1:=t; y:=2               [] x2=u ∧ y=1 → x2:=t; y:=2
           fi                                        fi
```

Figure 17: The two processes P_1 and P_2 of the synthesized 2-process concurrent program 2-*mutex*-4 of the form `x`$_1$`:=t; x`$_2$`:=t; y:=0; do` P_1 [] P_2 `od`. It enjoys the properties *ME*, *SF*, *BO*, and *MR*.

$$\begin{tikzcd}
 & & \langle t,w,2\rangle \arrow[ddl, "2.5"'] \arrow[ddr, "1.3"] & & & & & \langle w,t,2\rangle \arrow[ddl, "2.3"'] \arrow[ddr, "1.5"] & \\
 \arrow[r] & \langle t,t,0\rangle \arrow[ur, "2.1"] \arrow[urrrrrr, "1.1"] & & \langle w,u,1\rangle \arrow[urrrr, "2.7"] & & & \langle u,w,0\rangle \arrow[ullll, "1.7"'] & & \langle t,t,1\rangle \arrow[ullllll, "2.2"'] \arrow[ul, "1.2"'] \\
 & \langle t,u,2\rangle \arrow[u, "2.6"] \arrow[urr, "1.3"] & & \langle w,w,1\rangle \arrow[u, "2.4"'] & & & \langle w,w,0\rangle \arrow[u, "1.4"] & & \langle u,t,2\rangle \arrow[ull, "2.3"] \arrow[u, "1.6"']
\end{tikzcd}$$

Figure 18: The transition relation of the Kripke structure associated with the 2-process concurrent program 2-*mutex*-4 of Figure 17. An arch from s to t with label $i.n$ indicates that the guarded command (n) of process P_i (see Figure 17) causes the transition from state s to state t. The initial state is $\langle t,t,0\rangle$.

- *clasp* [66] (`http://potassco.sourceforge.net/`)
- *claspD* [53] (`http://potassco.sourceforge.net/`)
- *cmodels* [71] (`http://www.cs.utexas.edu/~tag/cmodels/`)
- *DLV* [99] (`http://www.dlvsystem.com/dlv/`)
- *GnT* [89] (`http://www.tcs.hut.fi/Software/gnt/`)
- *smodels* [134] (`http://www.tcs.hut.fi/Software/smodels/`)

The ground instances of Π given as input to the solvers *clasp*, *claspD*, *cmodels*, *GnT*, and *smodels*, have been generated by using *gringo* (`http://potassco.sourceforge.net/`). All experiments have been performed on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system. In order to compare the performance of the ASP solvers listed above, we have implemented the synthesis procedure by using the following encodings of Π_σ (that is, the program which generates the guarded commands): (i) Disjunctive Logic Program (denoted by Π_σ^{DLP}), (ii) Normal Logic Program (denoted by Π_σ^{NLP}), and (iii) Stratified Choice Integrity constraint Program (denoted by Π_σ^{SCI}) [116]. In particular, program Π_σ^{DLP} is a straightforward encoding of the logic program Π_σ of Definition 17, program Π_σ^{NLP} is derived from Π_σ by rewriting any rule of the form $a \vee b \leftarrow c$ as the pair of rules $a \leftarrow \texttt{not}\ b, c$ and $b \leftarrow \texttt{not}\ a, c$, and program Π_σ^{SCI} is derived from Π_σ by rewriting any rule of the form $a \vee b \leftarrow c$ as the choice rule [116] $\{a, b\} \leftarrow c$. We have executed each solver on each example using a 600 second timeout. All times required to generate the first solution and all solutions are reported in Tables 4.2, 4.3, and 4.4, for Π_σ^{DLP}, Π_σ^{NLP}, and Π_σ^{SCI}, respectively. Each table shows the results only for the ASP solvers which can to deal with the considered encoding.

The experimental results reported in Table 4.2 show that *claspD* is the ASP solver with the best performance on all synthesis examples obtained by using the Π_σ^{DLP} encoding. Regarding the other solvers we have that *cmodels* provides better timings than *GnT*, but the former crashes when exercised on Programs (2), (6), and (7) where *GnT* succeed.

Concerning the results obtained by using the Π_σ^{NLP} encoding (see Table 4.3), we observe that *clasp* and *claspD*, which is an extension of *clasp* for solving disjunctive logic programs, are the tools that perform better on almost all examples. We also have that they provide approximately the same performance on small instances (that is when the number of processes is small) of the synthesis problem (Programs (1) to (6)). However, the performance gap between *clasp* and *claspD* increases as the size of the instance of the synthesis problem

Table 4.2: Synthesis times using Π_σ^{DLP}. Numbers in column 'Program' refer to the synthesized programs listed in Table 4.1. Column 'First' and column 'All' show the time expressed in seconds to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{DLP} \cup \Pi_\varphi$. ∞ means 'no answer within 600 second'. e means 'tool crashes'.

Program	*claspD*		*cmodels*		*DLV*		*GnT*	
	First	All	First	All	First	All	First	All
(1)	0.01	0.01	0.01	0.02	0.01	0.02	0.02	0.04
(2)	0.01	0.01	0.02	e	0.01	0.09	0.02	0.05
(3)	0.03	0.03	0.06	0.08	0.21	1.18	0.12	0.18
(4)	0.03	0.05	0.07	0.09	0.16	3.19	0.13	0.23
(5)	0.14	0.17	0.26	0.57	∞	∞	0.84	5.92
(6)	0.04	0.05	e	e	0.40	∞	0.12	0.37
(7)	2.57	3.49	e	e	∞	∞	12.65	308.06
(8)	2.82	4.32	4.82	8.14	∞	∞	14.01	361.50
(9)	460.39	∞	544.03	∞	∞	∞	∞	∞
(10)	0.29	0.35	0.61	3.27	73.50	∞	1.17	3.65
(11)	2.07	2.89	3.10	106.04	∞	∞	10.90	71.22
(12)	12.39	20.43	18.37	∞	∞	∞	376.87	∞

increases (Programs (10) and (11)). Despite of worse timing results, *cmodels*, *GnT* and *smodels* succeed in synthesizing Program (12) where both *clasp* and *claspD* fail. Note also that, however, *claspD* is able to synthesize all solutions for Program (12) in 20.43 second if we consider the disjunctive logic program encoding.

Finally, concerning the results obtained by using Π_σ^{SCI} (see Table 4.4) we have that *clasp* is the ASP solver which performs better and, on large instances (Programs (7)–(12)) it outperforms *smodels*.

4.2 Related Work

Pioneering works on the synthesis of concurrent programs from temporal specifications are those by Clarke and Emerson [25] and Manna and Wolper [107]. In both these works the authors reduce the synthesis problem to the satisfiability problem of the given temporal specifications. Their synthesis methods exploit the finite model property for propositional temporal logics which asserts that if a given formula is satisfiable, then it is satisfiable in a finite model (whose size depends on the size of the formula).

In [25] Clarke and Emerson propose the following three-phase method for the synthesis of concurrent programs for a shared-memory model of execution: Phase 1 consists in providing the CTL specification of the concurrent program; Phase 2 consists in applying the tableau-based decision procedure for the satisfiability of CTL formulas to obtain a model of the CTL specification; and Phase 3 consists in extracting the synchronization skeletons from the model of the CTL specification.

Similarly, in [107] Manna and Wolper present a method that uses a tableau-based decision procedure for linear temporal logic (LTL) for the synthesis of synchronization instructions for processes in a message-passing model of execution.

However, the approaches proposed in [25, 107] have some drawbacks. In particular, they suffer from the state space explosion problem in that the models from which the synchronization instructions are extracted, have sizes which are exponential with respect to the number of processes. Moreover, the synthesized instructions work for models of computation which require further refinements for their use in a realistic architecture. Extensions of the synthesis methods of [25, 107] have been proposed by Attie and Emerson in [7] to deal with the state space explosion problem and allow an arbitrarily large number of processes by exploiting similarities among them. Also Attie and Emerson in [8] present an

Table 4.3: Synthesis times using Π_σ^{NLP}. Numbers in column ‘Program’ refer to the synthesized programs listed in Table 4.1. Column ‘First’ and column ‘All’ show the time expressed in seconds to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{NLP} \cup \Pi_\varphi$. ∞ means ‘no answer within 600 second’.

Program	*clasp*		*claspD*		*cmodels*		*DLV*		*GnT*		*smodels*	
	First	All	First	All	First	All	First	All	First	All	First	All
(1)	0.02	0.02	0.01	0.01	0.02	0.02	0.01	0.02	0.02	0.03	0.02	0.02
(2)	0.01	0.01	0.01	0.02	0.01	0.02	0.02	0.04	0.02	0.03	0.01	0.01
(3)	0.03	0.04	0.03	0.04	0.06	0.08	0.09	0.15	0.09	0.09	0.04	0.04
(4)	0.04	0.04	0.04	0.04	0.08	0.08	0.13	0.20	0.11	0.11	0.50	0.60
(5)	0.13	0.16	0.15	0.19	0.24	0.31	∞	∞	0.93	1.06	0.26	0.31
(6)	0.05	0.07	0.04	0.06	0.06	0.09	0.77	∞	0.12	0.17	0.07	0.09
(7)	2.11	3.08	3.34	4.38	2.94	12.21	∞	∞	32.90	36.51	5.56	18.86
(8)	4.90	6.23	3.30	6.29	7.18	18.90	∞	∞	89.79	96.79	12.76	61.22
(9)	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
(10)	0.29	1.39	0.31	0.70	0.47	0.74	126.21	∞	0.84	1.61	0.63	1.27
(11)	2.48	40.57	2.34	12.60	3.07	6.57	∞	∞	11.84	41.63	18.74	31.10
(12)	∞	∞	∞	∞	18.23	55.18	∞	∞	158.02	442.69	359.58	596.53

Table 4.4: Synthesis times using Π_σ^{SCI}. Numbers in column 'Program' refer to the synthesized programs listed in Table 4.1. Column 'First' and column 'All' show the time expressed in seconds to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{SCI} \cup \Pi_\varphi$. ∞ means 'no answer within 600 second'. $\perp$ means 'no models at all'. (*) means '*GnT* is able to generate one out of two models'. e means 'tool crashes'.

Program	*clasp*		*claspD*		*cmodels*		*GnT*		*smodels*	
	First	All	First	All	First	All	First	All	First	All
(1)	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.06	0.01	0.04
(2)	0.01	0.01	0.01	0.01	0.01	0.03	0.03	0.06	0.01	0.04
(3)	0.04	0.05	0.03	0.04	0.06	0.07	$\perp$	$\perp$	0.04	0.04
(4)	0.04	0.04	0.06	0.06	0.07	0.08	0.16	0.21	0.05	0.06
(5)	0.13	0.17	0.13	0.18	0.25	0.58	0.36	(*)	0.72	4.27
(6)	0.05	0.04	0.05	0.06	0.06	0.12	$\perp$	$\perp$	0.90	0.13
(7)	1.33	2.10	2.35	3.83	3.17	e	$\perp$	$\perp$	135.63	267.07
(8)	2.66	4.28	3.61	5.69	e	e	$\perp$	$\perp$	406.59	∞
(9)	444.14	∞	∞	∞	262.45	∞	∞	∞	∞	∞
(10)	0.30	0.34	0.30	1.70	0.48	3.20	1.60	4.49	0.95	1.59
(11)	2.40	2.89	2.25	26.45	3.19	106.34	22.15	91.42	20.76	105.15
(12)	22.62	25.61	18.71	406.76	19.28	∞	598.60	∞	499.96	∞

extension of their synthesis method to deal with a finer, more realistic atomicity of instructions so that only read and write operations are required to be atomic.

The papers we have considered so far refer to the synthesis of the so called closed systems, that is, the synthesis of programs whose processes are all specified by some given formulas. A different approach to the synthesis of concurrent programs has been presented by Pnueli and Rosner in [122]. These authors propose a method for synthesizing reactive modules of so called open systems, that is, systems in which the designer has no control over the inputs which come from an external environment. They introduce an automata-based synthesis procedure from a specification given as a linear temporal logic formula. The synthesis of open systems has also been studied by Kupferman and Vardi in [96]. Also the method they propose is based on automata-theoretical techniques. Paper [96] is important because it also presents some basic complexity results for the synthesis problems when specifications are given by CTL formulas or LTL formulas.

Our synthesis procedure follows the lines of [7, 25, 107] and considers concurrent programs to be closed systems. The advantage of our method resides in the fact that we solve the synthesis problem in a purely declarative manner. We reduce the problem of synthesizing a concurrent program to the problem of finding the answer sets of a logic program without the need for any *ad hoc* algorithm. Moreover, besides temporal properties, we can specify for the programs to be synthesized, some structural properties, such as various symmetry properties. Then, our ASP program automatically synthesizes concurrent programs which satisfy the desired properties. In order to reduce the search space when solving the synthesis problem, we have used the notion of symmetric concurrent programs which is similar to the one which was introduced in [7] to overcome the state space explosion problem. Our notion of symmetry is formalized within group theory, similarly to what has been done in [56] for the problem of model checking.

To the best of our knowledge, there is only one paper [80] by Heymans, Nieuwenborgh and Vermeir who make use, as we do, of Answer Set Programming for the synthesis of concurrent programs. The authors of [80] have extended the ASP paradigm by adding preferences among models and they have realized an answer set system, called OLPS. They perform the synthesis of concurrent programs following the approach proposed in [25] and, in particular, they use OLPS for Phase 2 of the synthesis procedure, having reduced the satisfiability problem of CTL formulas to the problem of constructing the answer sets of logic programs. The encoding proposed by [80] yields a synthesis procedure with

NEXPTIME time complexity and, thus, it is not optimal because the complexity of the problem of CTL satisfiability is EXPTIME [55].

On the contrary, our technique for reducing the satisfiability problem to the construction of the answer sets of logic programs, does not require any extension of the ASP paradigm. Indeed, we use standard ASP solvers, such as *claspD* [53], and every phase of our synthesis procedure is fully automatic. In particular, from any answer set we can mechanically derive the guarded commands which, by construction, guarantee that the synthesized program satisfies the given behavioural and structural properties. Moreover, we show that our method has optimal time complexity because it has EXPTIME complexity with respect to the size of the temporal specification.

CHAPTER 5

Proofs

We first introduce the following notions which will be used in the proofs.

A nonempty set I of ground atoms is *elementary* [67] for a program *ground*(Π) if for all nonempty proper subsets S of I there exists a rule r in *ground*(Π) such that: (i) $H(r) \cap S \neq \emptyset$, (ii) $B^{+}(r) \cap (I-S) \neq \emptyset$, (iii) $H(r) \cap (I-S) = \emptyset$, and (iv) $B^{+}(r) \cap S = \emptyset$. A program *ground*($\Pi$) is said to be *Head Elementary set Free* (HEF, for short) if, for every rule r in *ground*(Π), there is no elementary set Z for *ground*(Π) such that $|H(r) \cap Z| > 1$. We say that Π is HEF if *ground*(Π) is HEF. With any given HEF program Π we associate a normal logic program Π^n obtained from Π by replacing every rule r of Π of the form:

$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1} \wedge \ldots \wedge a_m \wedge \ \mathtt{not}\, a_{m+1} \wedge \ldots \wedge \mathtt{not}\, a_n$
for some $k > 1$, by the following k normal rules:

$a_j \leftarrow \bigwedge_{i\in\{1,\ldots,k\}-\{j\}} \mathtt{not}\, a_i \wedge a_{k+1} \wedge \ldots \wedge a_m \wedge \mathtt{not}\, a_{m+1} \wedge \ldots \wedge \mathtt{not}\, a_n$
for $j = 1, \ldots, k$. It can be shown that $ans(\Pi) = ans(\Pi^n)$ [67].

The following Proposition 3 is required for the proofs of Theorem 9 and Theorem 10.

Proposition 3. The logic program Π_σ is Head Elementary set Free.

Proof. We assume by contradiction that there exists a rule r in *ground*(Π_σ) and there exists a set Z which is an elementary set for *ground*(Π_σ) such that $|H(r) \cap Z| > 1$. If $|H(r) \cap Z| > 1$, then either:
(i) r is an instance of Rule 1.1 of Definition 17 and there exist $l \in \mathcal{L}$, $d \in \mathcal{D}$ such that

$\{enabled(1, l, d),\ disabled(1, l, d)\} \subseteq Z$, or
(ii) r is an instance of Rule 1.2 of Definition 17 and there exist $l, l', l'' \in \mathcal{L}$, $d, d', d'' \in \mathcal{D}$ such that

$\{gc(1,l,d,l',d'), gc(1,l,d,l'',d'')\} \subseteq Z$.
Let us consider Case (i). Let S be a nonempty proper subset of Z such that $\{enabled(1,l,d)\} \in S$ and $\{disabled(1,l,d)\} \notin S$. Clearly, $H(r) \cap (Z-S) \neq \emptyset$. This contradicts Condition (iii) for Z to be an elementary set for $ground(\Pi_\sigma)$.

Case (ii) is analogous to Case (i). Thus, we get that $ground(\Pi_\sigma)$ is HEF and, by definition, also Π_σ is HEF. □

By this proposition and the fact that the transformation from Π into Π^n presented above, preserves the answer set semantics when applied to HEF programs [67], we have that $ans(\Pi_\sigma) = ans(\Pi^n_\sigma)$, where program Π^n_σ is obtained from program Π_σ as follows:
(i) Rule 1.1 of program Π_σ is replaced by the following two normal rules:

$$enabled(1,X_1,Y) \leftarrow \texttt{not}\ disabled(1,X_1,Y) \wedge reachable(\langle X_1,\ldots,X_k,Y\rangle)$$
$$disabled(1,X_1,Y) \leftarrow \texttt{not}\ enabled(1,X_1,Y) \wedge reachable(\langle X_1,\ldots,X_k,Y\rangle),\text{ and}$$

(ii) Rule 1.2 of program Π_σ is replaced by m normal rules, for $i=1,\ldots,m$, each of which is of the form:

$$gc(1,X,Y,X_i,Y_i) \leftarrow \textstyle\bigwedge_{j\in\{1,\ldots,m\}-\{i\}} \texttt{not}\ gc(1,X,Y,X_j,Y_j) \wedge$$
$$enabled(1,X,Y) \wedge candidates(X,Y,[\langle X_1,Y_1\rangle,\ldots,\langle X_m,Y_m\rangle]).$$

From the fact that $\Pi_\varphi \rhd \Pi_\sigma$ and $ans(\Pi_\sigma) = ans(\Pi^n_\sigma)$, we get that (see end of Section 1.1):
$ans(\Pi) = ans(\Pi_\varphi \cup \Pi_\sigma) = \bigcup_{M\in ans(\Pi_\sigma)} ans(\Pi_\varphi \cup \overleftarrow{M}) = \bigcup_{M\in ans(\Pi^n_\sigma)} ans(\Pi_\varphi \cup \overleftarrow{M}) = ans(\Pi_\varphi \cup \Pi^n_\sigma)$.

Therefore, in order to compute all answer sets of program $\Pi_\varphi \cup \Pi_\sigma$, we can give $\Pi_\varphi \cup \Pi^n_\sigma$ as input to an answer set solver which does not support disjunctive logic programs.

Proof of Theorem 9

In order to prove Theorem 9 we first introduce the following Lemma A.2 stating the correctness of the logic program Π_φ (see Definition 18).

Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda\rangle$ be a Kripke structure. We introduce the program $P_\mathcal{K}$, called encoding of $\mathcal{K}$, consisting of $ground(\Pi_\varphi - \{\text{Rule 1}\}) \cup \{tr(s,t) \leftarrow \mid \langle s,t\rangle \in \mathcal{R}\}$. □

Note. $P_\mathcal{K}$ is a *locally stratified* normal program, thus it has a unique *stable model* [6] which coincides with its unique answer set $M|_{\{sat,satpath,elem,tr\}}$.

Lemma 1 (Correctness of Π_φ). Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be a Kripke structure and $P_\mathcal{K}$ be the encoding of $\mathcal{K}$. For all states $s \in \mathcal{S}$ and CTL formulas φ we have that: $\mathcal{K}, s \vDash \varphi$ iff $sat(s, \varphi) \in M(P_\mathcal{K})$ $\square$

Proof of A.2

The proof is by structural induction on φ. By induction hypothesis we assume that, for all states $s \in D$ and for all proper subformulas ψ of φ

$$\mathcal{K}, s \models \psi \ \text{ iff } \ sat(s, \psi) \in M(P_\mathcal{K}) \qquad (\dagger)$$

Now we consider the following cases.

Case 1. (φ is an elementary proposition e of the form $local(P_i, l)$ or of the form $shared(d)$)

For all states $s \in \mathcal{S}$ we have that:

$\mathcal{K}, s \models e$

iff $s(\mathtt{x}_i) = l$ (or $s(\mathtt{y}) = d$) (by Point (iv) of Def. 12)

iff $elem(s, e) \in M(P_\mathcal{K})$ (by Points (i) and (ii) of Def. 18 and by definition of $M(P_\mathcal{K})$)

iff $sat(s, e) \in M(P_\mathcal{K})$ (by Rule 2 of Π_φ and def. of $M(P_\mathcal{K})$)

Case 2. (φ is $\neg\psi$)

For all states $s \in \mathcal{S}$ we have that:

$\mathcal{K}, s \models \neg\psi$

iff $\mathcal{K}, s \models \psi$ does not hold (by def. of $\mathcal{K}, s \models \neg\psi$, see Sect. 1.1.3)

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

iff $sat(s, \neg\psi) \in M(P_\mathcal{K})$ ($M(P_\mathcal{K})$ is an answer set of $P_\mathcal{K}$)

Case 3. (φ is $\psi_1 \wedge \psi_2$)

For all states $s \in \mathcal{S}$ 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 def. of $\mathcal{K}, s \models \psi_1 \wedge \psi_2$, see Sect. 1.1.3)

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

iff $sat(s, \psi_1 \wedge \psi_2) \in M(P_\mathcal{K})$ (by Rule 4 of Π_φ and $M(P_\mathcal{K})$ is an answer set of $P_\mathcal{K}$)

Case 4. (φ is $\mathsf{EX}\,\psi$)

For all states $s \in \mathcal{S}$ we have that:

$\mathcal{K}, s \models \mathsf{EX}\,\psi$

iff there exists a state $s' \in \mathcal{S}$ such that
$\langle s, s' \rangle \in \mathcal{R}$ and $\mathcal{K}, s' \models \psi$ (by def. of $\mathcal{K}, s \vDash \mathsf{EX}\,\psi$, see Sect. 1.1.3)

iff there exists a state $s' \in \mathcal{S}$ such that:
(i) $tr(s, s') \in M(P_{\mathcal{K}})$, and
(ii) $sat(s', \psi) \in M(P_{\mathcal{K}})$ (by def. of $P_{\mathcal{K}}$ and (†))
iff there exist a state $s' \in \mathcal{S}$ and
a clause of the form $sat(s, ex(\psi)) \leftarrow t(s, s') \wedge sat(s', \psi)$ in $P_{\mathcal{K}}$
such that:
(i) $t(s, s') \in M(P_{\mathcal{K}})$, and
(ii) $sat(s', \psi) \in M(P_{\mathcal{K}})$ (by def. of $P_{\mathcal{K}}$)
iff $sat(s, ex(\psi)) \in M(P_{\mathcal{K}})$ (by Rule 5 of $P_{\mathcal{K}}$ and $M(P_{\mathcal{K}})$ is an answer set of $P_{\mathcal{K}}$)

Case 5. (φ is $\mathsf{EU}[\psi_1, \psi_2]$)
For any set Y of states, let $\tau_{\mathsf{EU}}(Y)$ denote the set $\{s \in \mathcal{S} \mid \mathcal{K}, s \models \psi_2\} \cup (\{s \in \mathcal{S} \mid \mathcal{K}, s \models \psi_1\} \cap \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}$ such that $\langle s, s'\rangle \in \mathcal{R}$ and $s' \in Y\})$. From [27] we have that $\mathcal{K}, s \models \mathsf{EU}[\psi_1, \psi_2]$ holds iff $s \in \mathit{lfp}(\tau_{\mathsf{EU}})$, where $\mathit{lfp}(\tau_{\mathsf{EU}}) = \tau^{\omega}_{\mathsf{EU}}$ Thus, we have to show that, for all states $s \in \mathcal{S}$, $s \in \mathit{lfp}(\tau_{\mathsf{EU}})$ iff $sat(s, eu(\psi_1, \psi_2)) \in M(P_{\mathcal{K}})$. Let P_{EU} be $\mathit{ground}(\{$Rule 6, Rule 7$\}) \cup \{sat(s, \psi_1) \in M(P_{\mathcal{K}}) \mid s \in \mathcal{S}\} \cup \{sat(s, \psi_2) \in M(P_{\mathcal{K}}) \mid s \in \mathcal{S}\} \cup \{tr(s, t) \in M(P_{\mathcal{K}}) \mid s, t \in \mathcal{S}\}$, and T^h_{EU} be the immediate consequence operator [6]. We proceed by induction on h: for all $h \geq 0$, for all $s \in \mathcal{S}$, $s \in \tau^h_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$. The base case trivially holds because $\tau^0_{\mathsf{EU}}(\emptyset) = \emptyset = \tau^h_{\mathsf{EU}}(\emptyset)$. Now, we assume the following inductive hypothesis:

for all $s \in \mathcal{S}$, $s \in \tau^h_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ (††)

and we prove that, for all $s \in \mathcal{S}$, $s \in \tau^{h+1}_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$.
We have that:
$s \in \tau^{h+1}_{\mathsf{EU}}(\emptyset)$
iff *either* $\mathcal{K}, s \models \psi_2$ (by def. of τ_{EU})
or $\mathcal{K}, s \models \psi_1$ and there exists a state $s' \in \mathcal{S}$ such that $\langle s, s'\rangle \in \mathcal{R}$ and $s' \in \tau^h_{\mathsf{EU}}(\emptyset)$ (by def. of τ_{EU})
iff *either* $sat(s, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ (by (††))
or $sat(s, \psi_1) \in T^h_{\mathsf{EU}}(\emptyset)$ and there exists a state $s' \in \mathcal{S}$ such that $\langle s, s'\rangle \in \mathcal{R}$ and $sat(s', eu(\psi_1, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ (by (††))
iff there exists a clause $\gamma \in P_{\mathcal{K}}$ such that:
either
(i) γ is of the form $sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_2)$, and
(ii) $sat(s, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ ($T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})
or there exists a state $s' \in \mathcal{S}$ such that:
(i) γ is of the form $sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_1) \wedge t(s, s') \wedge$

$sat(s', eu(\psi_1, \psi_2))$,
(ii) $sat(s, \psi_1) \in T^h_{\mathsf{EU}}(\emptyset)$,
(iii) $t(s, s') \in T^h_{\mathsf{EU}}(\emptyset)$, and
(iv) $sat(s', eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ (by definition of P_{EU} and $T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})

iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ ($T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})

Case 6. (φ is $\mathsf{EG}(\psi)$)
In order to prove this case we make use of the following Proposition A.3 [27]. Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be a Kripke structure and $\mathcal{K}' = \langle \mathcal{S}', \mathcal{S}'_0, \mathcal{R}', \lambda' \rangle$ be the Kripke structure obtained from $\mathcal{K}$ as follows: $\mathcal{S}' = \{s \in S \mid \mathcal{K}, s \vDash \psi\}$, $\mathcal{S}'_0 = \{s \in S_0 \mid \mathcal{K}, s \vDash \psi\}$, $\mathcal{R}' = \mathcal{R} \mid_{\mathcal{S}' \times \mathcal{S}'}$, and $\lambda' = \lambda \mid_{S'}$.

Proposition 4. $\mathcal{K}, s \vDash \mathsf{EG}\, \psi$ iff (i) $s_1 \in S'$, and (ii) there exists a path in $\mathcal{K}'$ that leads from s_1 to some node s_k in a nontrivial strongly connected component of the graph $\langle \mathcal{S}', \mathcal{R}' \rangle$.

Now, let us consider the program $P_{\mathcal{K}}$ constructed as indicated at the beginning of this section. We prove that conditions (i) and (ii) of Lemma 4 hold iff $sat(s, eg(\psi)) \in M(P_{\mathcal{K}})$:
$sat(s_1, eg(\psi)) \in M(P_{\mathcal{K}})$ iff (by Rule 8)
there exists a state $s_k \in S'$ such that
$\{\ satpath(s_1, s_k, \psi),\ satpath(s_k, s_k, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ iff (by def. of $\mathcal{P}_{\mathcal{K}}$)
$\{\ satpath(s_1, s_k, \psi) \leftarrow sat(s_1, \psi) \wedge tr(s_1, s_2) \wedge satpath(s_2, s_k, \psi)$,
$satpath(s_2, s_k, \psi) \leftarrow sat(s_2, \psi) \wedge tr(s_2, s_3) \wedge satpath(s_3, s_k, \psi)$,
$\ldots$
$satpath(s_{k-2}, s_k, \psi) \leftarrow sat(s_{k-2}, \psi) \wedge tr(s_{k-2}, s_{k-1}) \wedge$
$satpath(s_{k-1}, s_k, \psi)$,
$satpath(s_{k-1}, s_k, \psi) \leftarrow sat(s_{k-1}, \psi) \wedge tr(s_{k-1}, s_k)\ \} \subseteq P_{\mathcal{K}}$ iff
the following conditions holds:
(a) $\{\ sat(s_1, \psi), sat(s_2, \psi), \ldots, sat(s_{k-1}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(b) $\{\ tr(s_1, s_2), \ldots, tr(s_{k-1}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
(c) $\{\ satpath(s_k, s_k, \psi)\ \} \subseteq M(P_{\mathcal{K}'})$
iff the following conditions holds:
(a) $\{\ sat(s_1, \psi), sat(s_2, \psi), \ldots, sat(s_{k-1}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(b) $\{\ tr(s_1, s_2), \ldots, tr(s_{k-1}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
(c) $\{\ satpath(s_k, s_k, \psi) \leftarrow sat(s_k, \psi) \wedge tr(s_k, s_{k_1}) \wedge$
$satpath(s_{k_1}, s_k, \psi)$,
$satpath(s_{k_1}, s_k, \psi) \leftarrow sat(s_{k_1}, \psi) \wedge tr(s_{k_1}, s_{k_2}) \wedge$

$$\begin{array}{rl}
& \qquad satpath(s_{k_2}, s_k, \psi), \\
& \qquad \quad \dots \\
satpath(s_{k_{n-2}}, s_k, \psi) \leftarrow & sat(s_{k_{n-2}}, \psi) \wedge tr(s_{k_{n-2}}, s_{k_{n-1}}) \wedge \\
& \qquad satpath(s_{k_{n-1}}, s_n, \psi), \\
satpath(s_{k_{n-1}}, s_k, \psi) \leftarrow & sat(s_{k_{n-1}}, \psi) \wedge tr(s_{k_{n-1}}, s_k) \quad \} \subseteq P_{\mathcal{K}}
\end{array}$$

iff the following conditions holds:

(a) $\{sat(s_1, \psi), sat(s_2, \psi), \dots, sat(s_{k-1}, \psi)\} \subseteq M(P_{\mathcal{K}})$ (by (†))

(b) $\{tr(s_1, s_2), \dots, tr(s_{k-1}, s_k)\} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)

(c_1) $\{sat(s_{k_1}, \psi), sat(s_{k_2}, \psi), \dots, sat(s_{k_{n-1}}, \psi)\} \subseteq M(P_{\mathcal{K}})$ (by (†))

(c_2) $\{tr(s_k, s_{k_1}), tr(s_{k_1}, s_{k_2}), \dots, tr(s_{k_{n-1}}, s_k)\} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)

iff there exists a path $\pi = s_1, \dots, s_k$ of length $k \geq 1$ and s_k belongs to a nontrivial strongly connected component of $\langle \mathcal{S}', \mathcal{R}' \rangle$. □

Now we prove Theorem 9.

Let Π be the program $\Pi_\varphi \cup \Pi_\sigma$. We need the following notation. Given a set P of predicate symbols and a set M of atoms, we define $M|_P$ to be the set $\{A \in M \mid$ the predicate of A is in $P\}$.

(*if.* Soundness) Let M be an answer set of Π. Recall that σ is of the form $\langle f, T, l_0, d_0 \rangle$. Let us consider a command C of the form: $\mathtt{x}_1 := l_0; \dots; \mathtt{x}_k := l_0;\ \mathtt{y} := d_0;$ `do` P_1 $[\!]$ $\dots$ $[\!]$ P_k `od`, where for $i = 1, \dots, k$, $(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d')$ is in P_i iff $gc(i, l, d, l', d') \in M$.

We have the following two properties of C.

(CP1) For $i = 1, \dots, k$, every guarded command in P_i is of the form $\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l';\ \mathtt{y} := d'$ with $\langle l, d \rangle \neq \langle l', d' \rangle$. Indeed, M is a model of Π_σ and, in particular, of the ground facts defining the predicate *candidates* (see Definition 17).

(CP2) For $i = 1, \dots, k$, the guards of any two guarded commands of process P_i are mutually exclusive. Indeed, the following holds. By Proposition 3, Π_σ is HEF. Hence, by Rule 2.1, for every $l \in \mathcal{L}$ and $d \in \mathcal{D}$, at most one atom of the form $gc(1, l, d, l', d')$ belongs to M. Since M is a supported model [21], by Rule 2.i, for $i = 2, \dots, k$, we get that $gc(i, l, f(d), l', f(d')) \in M$ iff $gc(i-1, l, d, l', d') \in M$. By using this fact we get that, for $i = 1, \dots, k$, for every $l \in \mathcal{L}$ and $d \in \mathcal{D}$, at most one atom of the form $gc(i, l, d, l', d')$ belongs to M.

By Properties (CP1) and (CP2), C is a k-process concurrent program (see Definition 10).

Now, we prove that: (i) C satisfies φ and (ii) C is symmetric w.r.t. σ.

Point (i). Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be the Kripke structure associated with C, constructed as indicated in Definition 12 and $P_{\mathcal{K}}$ be the encoding of $\mathcal{K}$. We have that:

M is an answer set of Π

iff M is an answer set of $\Pi - \{\leftarrow \texttt{not}\ sat(s_0, \varphi)\}$ and $sat(s_0, \varphi) \in M$

(by def. of integrity constraint)

iff $M = M(\Pi_\varphi - \{\leftarrow \texttt{not}\ sat(s_0, \varphi)\} \cup$
$M\ |_{\{gc, enabled, disabled, reachable, tr, candidates, perm\}})$

(by def. of $\Pi_\varphi \cup \Pi_\sigma$)

iff $M = M(\mathcal{P}_k) \cup M\ |_{\{gc, enabled, disabled, reachable, tr, candidates, perm\}})$

(by def. of $P_{\mathcal{K}}$)

iff $M\ |_{\{sat, satpath, tr, elem\}} = M(\mathcal{P}_k)$.

Since $sat(s, \varphi) \in M$, we obtain that $sat(s, \varphi) \in M(\mathcal{P}_k)$ and, thus, $\mathcal{K}, s \vDash \varphi$.

Point (ii). By construction, C is of the form $\texttt{x}_1 := l_0; \ldots; \texttt{x}_k := l_0; \texttt{y} := d_0;$ $\texttt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \texttt{od}$. Let us now prove that Conditions (i) and (ii) of Definition 15 hold.

For all $gc(i, l, d, l', d') \in M$ we have that the pair $\langle l', d' \rangle$ belongs to the list L which is the third argument of $candidates(l, d, L)$. By Point (i) of Definition 17, for every pair $\langle l', d' \rangle$ in L we have that $\langle l, l' \rangle \in T$ and, therefore, C satisfies Condition (i) of Definition 15.

Since M is a supported model of $ground(\Pi)^M$ and Rule 1.i, for $1 < i \leq k$, is the only rule in Π whose head is unifiable with $gc(i, l, d, l', d')$ we have that $gc(i-1, l, d, l', d') \in M$ iff $gc(i, l, f(d), l', f(d')) \in M$. Thus, Condition (ii) of Definition 15 holds for C because f is a permutation of order k.

(*only if.* Completeness) Let C be a k-process concurrent program which satisfies φ and is symmetric w.r.t. σ, and $\mathcal{K}$ be the Kripke structure $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ associated with C whose processes are $P_1, \ldots, P_k$. We have to prove that there exists an answer set $M \in ans(\Pi_\varphi \cup \Pi_\sigma)$ which encodes C. Let M be defined as follows.

$$
\begin{array}{rll}
M = & \{reachable(s) \mid s \in \mathcal{S}\} & (M.1)\\
\cup & \{tr(s, s') \mid \langle s, s' \rangle \in \mathcal{R}\} & (M.2)\\
\cup & \{gc(i, l, d, l', d') \mid (\texttt{x}_i = l \land \texttt{y} = d \rightarrow \texttt{x}_i := l'; \texttt{y} := d') & \\
 & \qquad \text{is in } P_i \land 1 \leq i \leq k\} & (M.3)\\
\cup & \{enabled(i, l, d) \mid \exists\, l', d'\ (\texttt{x}_i = l \land \texttt{y} = d \rightarrow \texttt{x}_i := l'; \texttt{y} := d') &
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad\qquad\text{is in } P_i \wedge 1 \leq i \leq k\} && (M.4)\\
\cup\ & \{\mathit{disabled}(1, s(\mathtt{x}_1), s(\mathtt{y})) \mid s \in \mathcal{S} \wedge \\
&\qquad\qquad \neg\exists\, c\ (\mathtt{x}_1 = s(\mathtt{x}_1) \wedge \mathtt{y} = s(\mathtt{y}) \rightarrow c) \text{ is in } P_1\} && (M.5)\\
\cup\ & \{\mathit{sat}(s, \psi) \mid s \in \mathcal{S} \wedge \mathcal{K}, s \vDash \psi\} && (M.6)\\
\cup\ & \{\mathit{satpath}(s_0, s_n, \psi) \mid \exists\, \langle s_0, \ldots, s_n\rangle \\
&\qquad\qquad \forall i\ (0 \leq i \leq n \rightarrow \mathcal{K}, s_i \vDash \psi)\ \} && (M.7)\\
\cup\ & \{\mathit{elem}(p, s) \mid s \in \mathcal{S} \wedge p \in \lambda(s)\} && (M.8)\\
\cup\ & \{\mathit{perm}(d, d') \mid d, d' \in \mathcal{D} \wedge f(d) = d'\} && (M.9)\\
\cup\ & \{\mathit{candidates}(l, d, L(l, d)) \leftarrow \mid\ l \in \mathcal{L} \wedge d \in \mathcal{D}\} && (M.10)
\end{aligned}
$$

where $L(l, d)$ is any list representing the set $\{\langle l', d'\rangle \mid \langle l, l'\rangle \in T \ \wedge\ d' \in \mathcal{D} \ \wedge\ \langle l, d\rangle \neq \langle l', d'\rangle\}$ of pairs.
By $M.3$ and Definition 4.4 we have that M encodes C. Now we prove that M is an answer set of Π, that is, (i) M is a model of $\mathit{ground}(\Pi_\varphi \cup \Pi_\sigma)^M$ and (ii) M is a minimal such model.

(i) We prove that for every rule $r \in \mathit{ground}(\Pi_\varphi \cup \Pi_\sigma)^M$ if $B^+(r) \subseteq M$ then $H(r) \cap M \neq \emptyset$. We proceed by cases. Let us first consider the rules in $\mathit{ground}(\Pi_\sigma)$.

(Rule 1.1) Assume that r is $\mathit{enabled}(1, l_1, d) \vee \mathit{disabled}(1, l_1, d) \leftarrow \mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle)$. If $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \in M$ then, by $M.1$, we have that $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$. Since $\mathcal{R}$ is a total relation, either P_1 is enabled in $\langle l_1, \ldots, l_k, d\rangle$ and consequently, by $M.4$, $\mathit{enabled}(1, l_1, d) \in M$, or it is not enabled and thus, by $M.5$, $\mathit{disabled}(1, l_1, d) \in M$.

(Rule 1.i, $i > 1$) Assume that r is $\mathit{enabled}(i, l, d) \leftarrow \mathit{gc}(i, l, d, l', d')$. If $\mathit{gc}(i, l, d, l', d') \in M$ then, by $M.3$, $(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d')$ is in P_i, and consequently, by $M.4$, $\mathit{enabled}(i, l, d) \in M$.

(Rule 2.1) Assume that r is $\mathit{gc}(1, l, d, l_1, d_1) \vee \ldots \vee \mathit{gc}(1, l, d, l_m, d_m) \leftarrow \mathit{enabled}(1, l, d) \wedge\ \mathit{candidates}(l, d, [\langle l_1, d_1\rangle, \ldots, \langle l_m, d_m\rangle])$ for some $m \geq 1$. If $\mathit{enabled}(1, l, d) \in M$ then, by $M.4$, there exists in P_1 a guarded command whose guard is $\mathtt{x}_1 = l \wedge \mathtt{y} = d$ and the associated command is encoded as a pair $\langle l', d'\rangle$ occurring in the third argument of $\mathit{candidates}(l, d, [\langle l_1, d_1\rangle, \ldots, \langle l_m, d_m\rangle])$. Hence, by $M.3$, we have that $\mathit{gc}(1, l, d, l', d') \in M$.

(Rule 2.i, $i > 1$) Assume that r is $\mathit{gc}(i, l, e, l', e') \leftarrow \mathit{gc}(i{-}1, l, d, l', d') \wedge \mathit{perm}(d, e) \wedge \mathit{perm}(d', e')$, with $i > 1$. By Definition 15 we have that $(\mathtt{x}_i = l \wedge \mathtt{y} = f(d) \rightarrow \mathtt{x}_i := l'; \mathtt{y} := f(d'))$ is in P_i iff $(\mathtt{x}_{i-1} = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_{i-1} := l'; \mathtt{y} := d')$ is in P_{i-1} and, therefore, if $\mathit{gc}(i-1, l, d, l', d') \in M$, $f(d) = e$, and $f(d') = e'$ then, by $M.3$, $\mathit{gc}(i, l, e, l', e') \in M$.

(Rule 3.1) Assume that r is $\mathit{reachable}(s_0) \leftarrow$. Since $s_0 \in \mathcal{S}$, we have that by $M.1$, $\mathit{reachable}(s_0) \in M$.

(Rule 3.2) Assume that r is $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \leftarrow \mathit{tr}(\langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle)$. If we have that $\mathit{tr}(\langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle) \in M$ then, by $M.2$, $\langle\langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle\rangle \in \mathcal{R}$. Thus, $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$ and consequently, by $M.1$, $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \in M$.

(Rule 4.1–4.k) Assume that r is $\mathit{tr}(s,t) \leftarrow \mathit{reachable}(s) \wedge \mathit{gc}(i, l, d, l', d')$, with $s(\mathtt{x}_i) = l$, $s(\mathtt{y}) = d$, $t(\mathtt{x}_i) = l'$, and $t(\mathtt{y}) = d'$. If $\{\mathit{reachable}(s), \mathit{gc}(i, l, d, l', d')\} \subseteq M$ then $s \in \mathcal{S}$ and there exists a guarded command of the form $(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d')$ in P_i. Thus, by Definition 11, $\langle s, t\rangle \in \mathcal{R}$ and consequently, by $M.2$, we get that $\mathit{tr}(s,t) \in M$.

(Rule 5) Assume that r is $\leftarrow \mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle)$. We show that $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \notin M$. Let us assume, by contradiction, that $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \in M$ and, thus, by M.1, $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$. Since $\mathcal{R}$ is total, for every reachable state s, there exists a process P_i which is enabled in s, that is, by $M.4$, $\mathit{enabled}(i, l_i, d) \in M$, contradicting the hypothesis that $r \in \mathit{ground}(\Pi_\sigma)^M$, that is, for all $i \in \{1, \ldots, k\}$, $\mathit{enabled}(i, l_i, d) \notin M$.

Now we consider the rules in $\mathit{ground}(\Pi_\varphi)$.

(Rule 1) Since C satisfies φ, by $M.6$ we have that $\mathit{sat}(s_0, \varphi) \in M$ and, hence, $\{\leftarrow \ \mathtt{not}\ \mathit{sat}(s_0, \varphi)\}^M = \emptyset$. Thus, no rule of $\mathit{ground}(\Pi_\varphi)^M$ is obtained from Rule 1 by the Gelfond-Lifschitz transformation.

(Rules 2-8) Let $P_\mathcal{K}$ be the encoding of $\mathcal{K}$. By definition $M.6 = \{\mathit{sat}(s, \psi) \mid s \in \mathcal{S} \wedge \mathcal{K}, s \vDash \psi\}$. Moreover, by Lemma 1 we have that, for all $s \in \mathcal{S}$ and CTL formulas ψ, if $\mathcal{K}, s \vDash \psi$ then $\mathit{sat}(s, \psi) \in M(P_\mathcal{K})$. Thus, $M.2 \cup M.6 \cup M.7 \cup M.8 = M(P_\mathcal{K})$ is a model of Rules 2-8.

(Rule 9) Assume that r is $\mathit{satpath}(s, t, \psi) \leftarrow \mathit{sat}(s, \psi) \wedge \mathit{tr}(s,t)$. Assume that $\{\mathit{sat}(s, \psi), \mathit{tr}(s,t)\} \subseteq M$. Then, $\mathcal{K}, s \vDash \psi$ and $\langle s, t\rangle \in \mathcal{R}$ hold. Hence, by $M.7$, we have that $\mathit{satpath}(s, t, \psi) \in M$.

(Rule 10) Assume that r is $\mathit{satpath}(u_0, u_n, \psi) \leftarrow \mathit{sat}(u_0, \psi) \wedge \mathit{tr}(u_0, u_1) \wedge \mathit{satpath}(u_1, u_n, \psi)$. Assume that $\{\mathit{sat}(u_0, \psi), \mathit{tr}(u_0, u_1), \mathit{satpath}(u_1, u_n, \psi)\} \subseteq M$. Then, $\mathcal{K}, u_0 \vDash \psi$, $\langle u_0, u_1\rangle \in \mathcal{R}$, and there exists a finite path $\langle u_1, \ldots, u_n\rangle$, with $n > 1$, such that for all $1 \leq i \leq n$, $\mathcal{K}, u_i \vDash \psi$. Thus, by $M.7$, $\mathit{satpath}(u_0, u_n, \psi) \in M$.

(ii) We have to prove that M is a minimal (w.r.t. set inclusion) model of $\mathit{ground}(\Pi)^M$. We prove it by contradiction. Let us assume that M' is a model of $\mathit{ground}(\Pi)^M$ such that $M' \subset M$. Let z be a ground atom in $M - M'$. We proceed by cases.

(Case A) Assume that z is $\mathit{gc}(i,l,d,l',d')$. Thus, by $M.3$, there exists a guarded command in C whose encoding does not belong to M', and consequently, M' does not encode C.

(Case B) For every $s \in \mathcal{S}$, we define $h(s)$ to be the least integer $k \geq 0$ such that $\mathit{Reach}^k(s_0, s)$ holds. Assume that z is $\mathit{reachable}(s)$. Without loss of generality, we may assume that s is a state such that $\forall r \in \mathcal{S}$ if $\mathit{reachable}(r) \in M - M'$, then $h(r) \geq h(s)$. We have the following two cases.
(Case B.1) $s = s_0$. We get a contradiction from the fact that M' is a model of $\mathit{ground}(\Pi)^M$ and, thus, M' satisfies Rule 3.1.
(Case B.2) $s \neq s_0$. We have that there exists no $t \in \mathcal{S}$ such that $\mathit{tr}(t,s) \in M'$ (otherwise, since M' satisfies Rule 3.2, we would have $\mathit{reachable}(s) \in M'$). Take any $t \in \mathcal{S}$ such that $\mathit{Reach}^{h(s)-1}(s_0, t)$. Since M' satisfies Rules 4.1–4.k and $\mathit{tr}(t,s) \notin M'$, one of the following two facts holds.
Either (B.2.1) $\mathit{reachable}(t) \notin M'$. By M.1 we have that $\mathit{reachable}(t) \in M$, and thus, $\mathit{reachable}(t) \in M - M'$. Since $h(t) < h(s)$, we get a contradiction with the assumption that $\forall r \in \mathcal{S}$ if $\mathit{reachable}(r) \in M - M'$, then $h(r) \geq h(s)$.
Or (B.2.2) there exists no process i such that $\mathit{gc}(i, t(\mathtt{x}_i), t(\mathtt{y}), s(\mathtt{x}_i), s(\mathtt{y})) \in M'$. Therefore, the proof proceeds as in Case (A).

(Case C) Assume that z is $\mathit{enabled}(i,l,d)$. Since M' satisfies Rule 1.i, there exist no l' and d', such that $\mathit{gc}(i,l,d,l',d') \in M'$. Therefore, the proof proceeds as in Case (A).

(Case D) Assume that z is $\mathit{disabled}(1,l,d)$. By M.4 and M.5, we have that $\mathit{enabled}(1,l,d) \notin M$. Since M' satisfies Rule 1.1, one of the following two facts hold.
Either (D.1) No atom of the form $\mathit{reachable}(\langle l, l_2, \ldots, l_k, d\rangle)$ belongs to M'. Therefore, the proof proceeds as in Case (B).
Or (D.2) $\mathit{enabled}(1,l,d)$ belongs to M'. Therefore, we get a contradiction with the facts that $M' \subset M$ and $\mathit{enabled}(1,l,d) \notin M$.

(Case E) Assume that z is $\mathit{tr}(t,s)$. Since M' satisfies Rules 4.1–4.k, one of the following two facts hold.
Either (E.1) $\mathit{reachable}(t) \notin M'$. Therefore, the proof proceeds as in Case (B).
Or (E.2) There is no process i such that $\mathit{gc}(i, t(\mathtt{x}_i), t(\mathtt{y}), s(\mathtt{x}_i), s(\mathtt{y})) \in M'$. Therefore, the proof proceeds as in Case (A).

(Case F) Assume that z has one of these forms: $\mathit{sat}(s,\psi)$, or $\mathit{satpath}(s,t,\psi)$, or $\mathit{elem}(s,p)$. By $M.6$, $M.7$, $M.8$, and Lemma 1, we have that $M|_{\{\mathit{sat},\mathit{satpath},\mathit{elem},\mathit{tr}\}}$ is the least Herbrand model of $\mathit{ground}(\Pi_\varphi)^M \cup \overleftarrow{M|_{\{\mathit{tr}\}}}$. Now, since M' is an

Herbrand model of $ground(\Pi_\varphi)^M \cup \overleftarrow{M|_{\{tr\}}}$, we get that $M|_{\{sat,satpath,elem,tr\}} \subseteq M'$, thereby contradicting the assumption that $z \in M - M'$. $\square$

Proof of Theorem 10

Let $|ground(\Pi)|$ denote the size (that is, the number of rules) of $ground(\Pi)$. We have that $|ground(\Pi)|$ is $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$, where $k > 1$. Moreover, since program Π_σ is an HEF (see Proposition 3) logic program, Π_σ can be transformed into a normal logic program Π^n_σ such that $ans(\Pi_\sigma) = ans(\Pi^n_\sigma)$. We have that $|ground(\Pi^n_\sigma)| = \alpha_1 + \alpha_2 + |ground(\Pi_\sigma)|$, where α_1 depends on the number of the ground instances of Rule 1.1 and α_2 depends on the number of the ground instances of Rule 2.1. Now we have that: (i) α_1 is at most $|\mathcal{L}|^k \cdot |\mathcal{D}|$ (indeed, the ground instances of Rule 1.1 are at most $|\mathcal{L}|^k \cdot |\mathcal{D}|$), and (ii) α_2 is $\mathcal{O}(|\mathcal{L}|^2 \cdot |\mathcal{D}|^2)$ (indeed, the ground instances of Rule 2.1 are at most $|\mathcal{L}| \cdot |\mathcal{D}|$, and in any instance of Rule 2.1 the value of m is at most $|\mathcal{L}| \cdot |\mathcal{D}|$). Thus, $\alpha_1 + \alpha_2$ is $\mathcal{O}(|\mathcal{L}|^k \cdot |\mathcal{D}|^2)$ and $|ground(\Pi^n)|$ is $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$.

Given a set I of ground atoms, (i) to compute $ground(\Pi^n)^I$ takes linear time w.r.t. $|ground(\Pi^n)|$, (ii) to generate the minimal model M of $ground(\Pi^n)^I$ takes linear time w.r.t. $|ground(\Pi^n)^I|$, and (iii) to check whether or not $I = M$ also takes linear time w.r.t. $|ground(\Pi^n)^I|$ (for more information on these results the reader may refer to [129]). Hence, to verify whether or not a given set of ground atoms is an answer set of Π takes linear time w.r.t. $|ground(\Pi^n)|$. Thus, the verification that I is an answer set of Π takes exponential time w.r.t. k, linear time w.r.t. $|\varphi|$, and polynomial time w.r.t. $\mathcal{L}$ and w.r.t. $\mathcal{D}$.

Now, the choice of a candidate answer set I can be done by: (i) choosing, for each $\langle l, d\rangle \in \mathcal{L} \times \mathcal{D}$, at most one ground atom in the set $\{gc(1, l, d, l', d') \mid \langle l, l'\rangle \in T \wedge d' \in \mathcal{D} \wedge \langle l, d\rangle \neq \langle l', d'\rangle\}$, (ii) computing in $\mathcal{O}(k)$ time a ground atom of the form $gc(i, \ldots)$, for $i = 2, \ldots, k$, (iii) computing in $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$ time the ground instances of the rules in Π, where the truth values of the gc atoms are fixed as indicated at Steps (i) and (ii), thereby obtaining a stratified program, and (iv) finally, computing in $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$ the unique stable model of that stratified program.

Since Step (i) can be done in nondeterministic polynomial time w.r.t. $|\mathcal{L}| \times |\mathcal{D}|$, we get the thesis. $\square$

CHAPTER 6

Source code

In this section we list the source code used to synthesize 2-*mutex*-1 with *claspD* (Program (2) in Table 4.1).

The disjunctive logic program Π_σ^{DLP} encoding a structural property

This program is the encoding of the program Π_σ of the Definition 17, for $k = 2$ where rules 2.1–2.2 have been obtained by unfolding the definition of *candidates* and *perm*.

```
enabled(1,X1,Y) | disabled(1,X1,Y):- reachable(X1,X2,Y).
enabled(2,X2,Y) :- gc(2,X2,Y,X2p,Yp).

gc(1,t,Y,w,0) | gc(1,t,Y,w,1) | gc(1,t,Y,w,2):-
    enabled(1,t,Y).
gc(1,w,0,w,1) | gc(1,w,0,w,2) |
gc(1,w,0,u,0) | gc(1,w,0,u,1) | gc(1,w,0,u,2):-
    enabled(1,w,0).
gc(1,w,1,w,0) | gc(1,w,1,w,2) |
gc(1,w,1,u,0) | gc(1,w,1,u,1) | gc(1,w,1,u,2):-
    enabled(1,w,1).
gc(1,w,2,w,0) | gc(1,w,2,w,1) |
gc(1,w,2,u,0) | gc(1,w,2,u,1) | gc(1,w,2,u,2) -
    enabled(1,w,2).
```

```
gc(1,u,Y,t,0) | gc(1,u,Y,t,1) | gc(1,u,Y,t,2):-
    enabled(1,u,Y).
gc(2,X2,Z,X2p,Zp) :- gc(1,X2,Y,X2p,Yp), perm(Y,Z),
    perm(Yp,Zp).

reachable(X1,X2,Y) :- s0(X1,X2,Y).
reachable(X1p,X2p,Yp) :- reachable(X1,X2,Y),
    tr(X1,X2,Y,X1p,X2p,Yp).

tr(X1,X2,Y,X1p,X2,Yp):- reachable(X1,X2,Y),
    gc(1,X1,Y,X1p,Yp).
tr(X1,X2,Y,X1,X2p,Yp):- reachable(X1,X2,Y),
    gc(2,X2,Y,X2p,Yp).

 :- reachable(X1,X2,Y), not enabled(1,X1,Y), not
     enabled(2,X2,Y).
```

where: (i) variables $\texttt{X1}, \texttt{X2}, \texttt{X1p}$, and $\texttt{X2p}$ range over $\mathcal{L} = \{\texttt{t}, \texttt{w}, \texttt{u}\}$, (ii) variables $\texttt{Y}, \texttt{Yp}$, and $\texttt{Z}$ range over $\mathcal{L} = \{0, 1, 2\}$, (iii) the 2-generating function $f_2 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 2, 2\rangle\}$ is encoded by the following facts `perm(0,1)`, `perm(1,0)` and `perm(2,2)`. (iv) the initial state is encoded by $\texttt{s0}(\texttt{t}, \texttt{t}, 0)$. Note that the *local transition relation* $T = \{\langle \texttt{t}, \texttt{w}\rangle, \langle \texttt{w}, \texttt{w}\rangle, \langle \texttt{w}, \texttt{u}\rangle, \langle \texttt{u}, \texttt{t}\rangle\}$ is directly embedded into the guarded commands for process P_1.

The logic program Π_φ encoding a behavioural property

This program is the encoding of the program Π_σ of the Definition 18.

```
sat(X1,X2,Y,local(p1,X1)) :- elem(local(p1,X1),X1,X2,Y).
sat(X1,X2,Y,local(p2,X2)) :- elem(local(p2,X2),X1,X2,Y).
sat(X1,X2,Y,shared(Y)) :- elem(shared(Y),X1,X2,Y).
sat(X1,X2,Y,n(F)) :- not sat(X1,X2,Y,F), dp(n(F)), l(X1),
    l(X2), d(Y).
sat(X1,X2,Y,o(F,G)) :- sat(X1,X2,Y,F), dp(o(F,G)).
sat(X1,X2,Y,o(F,G)) :- sat(X1,X2,Y,G), dp(o(F,G)).
sat(X1,X2,Y,a(F,G)) :- sat(X1,X2,Y,F), sat(X1,X2,Y,G),
    dp(a(F,G)).
sat(X1,X2,Y,ex(F)) :- tr(X1,X2,Y,X1p,X2p,Yp),
    sat(X1p,X2p,Yp,F), dp(ex(F)).
```

```
 9  sat(X1,X2,Y,eu(F,G)) :- sat(X1,X2,Y,G), dp(eu(F,G)).
10  sat(X1,X2,Y,eu(F,G)) :- sat(X1,X2,Y,F),
        tr(X1,X2,Y,X1p,X2p,Y), sat(X1p,X2p,Y,eu(F,G)),
        dp(eu(F,G)).
11  sat(X1,X2,Y,ef(F)) :- sat(X1,X2,Y,F), dp(ef(F)).
12  sat(X1,X2,Y,ef(F)) :- tr(X1,X2,Y,X1p,X2p,Yp),
        sat(X1p,X2p,Yp,ef(F)), dp(ef(F)).
13  sat(X1,X2,Y,eg(F)) :- satpath(X1,X2,Y,X1p,X2p,Yp,F),
        satpath(X1p,X2p,Yp,X1p,X2p,Yp,F), dp(eg(F)).
14  satpath(X1,X2,Y,X1p,X2p,Yp,F) :- sat(X1,X2,Y,F),
        tr(X1,X2,Y,X1p,X2p,Yp), dp(eg(F)).
15  satpath(X1,X2,Y,X1p,X2p,Yp,F) :- sat(X1,X2,Y,F),
        tr(X1,X2,Y,X1pp,X2pp,Ypp),
        satpath(X1pp,X2pp,Ypp,X1p,X2p,Yp,F), dp(eg(F)).
```

where: (i) variables X1, X1p, X1pp, X2, and X2p range over $\mathcal{L} = \{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, (ii) variables Y, Yp, and Z range over $\mathcal{L} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$, (iii) variables F and G range over the term encoding the formula φ and its subterms, and (iv) predicate dp encodes the domain predicate which restricts the domain of variables F and G.

As an example we list: (i) the domain predicates (lines 1–5), (ii) the elementary properties (lines 8–9), and (iii) the integrity constraint (line 10) needed to ensure that processes P_1 and P_2, obtained by using $\Pi = \Pi_\sigma^{DLP} \cup \Pi_\varphi$, enjoy the mutual execution property.

```
 1  dp(n(ef(n(n(a(local(p1,u),local(p2,u))))))).
 2  dp(ef(n(n(a(local(p1,u),local(p2,u)))))).
 3  dp(n(n(a(local(p1,u),local(p2,u))))).
 4  dp(n(a(local(p1,u),local(p2,u)))).
 5  dp(a(local(p1,u),local(p2,u))).
 6
 7  elem(local(p1,u),u,X2,Y) :- l(X2), d(Y).
 8  elem(local(p2,u),X1,u,Y) :- l(X1), d(Y).
 9
10  :- not
        sat(X1,X2,Y,n(ef(n(n(a(local(p1,u),local(p2,u))))))),
        s0(X1,X2,Y).
```

Note that the integrity constraint at line 10 is the encoding or Rule 1 of the Definition 18

Conclusions

The first contribution of this thesis concerns the design of a general *verification framework* and its implementation to perform *software model checking* of imperative programs.

We have considered *program transformation* and *constraint logic programming* (or, equivalently, *constrained Horn clauses*) as foundational building blocks to define the verification framework.

Although the approach based on CLP program transformation shares many ideas and techniques with abstract interpretation and automated theorem proving, we believe that it has some distinctive features that make it quite appealing.

Let us point out some advantages of the techniques for software model checking which, like ours, use methodologies based on program transformation and constraint logic programming.

1. The approach is parametric with respect to the program syntax and semantics, because interpreters and proof systems can easily be written in CLP, and verification conditions can automatically be generated by specialization. Thus, transformation-based verifiers can be easily adapted to the changes of the syntax and the semantics of the programming languages under consideration and also to the different logics where the properties of interest are expressed.
2. Program transformation provides a uniform framework for program analysis. Indeed, abstraction operators can be regarded as particular generalization operators and, moreover, transformation-based verification can be easily combined to other program transformation techniques, such as program slicing, dead code elimination, continuation passing transformation, and loop fusion.
3. By applying suitable generalization operators we can guarantee that transformation always terminates and produces an equivalent program with re-

spect to the property of interest. Thus, we can apply a sequence of transformations, thereby refining the analysis to the desired degree of precision.

Indeed, in this thesis we have shown that one can construct a framework where the generation of *verification conditions* and their verification can both be viewed as program transformations.

Let us summarize here the main contributions of this thesis with regard to program verification.

(i) The adaptation and the integration of various techniques for specializing and transforming constraint logic programs into a general verification framework which is parametric with respect to the various syntaxes and semantics of the programming languages under consideration and also to the various logics in which the properties of interest may be expressed.

    This adaptation has required the customization of general purpose unfolding and generalization strategies to the specific task of specializing the interpreter of the programming language under consideration, as well as the programs derived from the interpreter in subsequent iterations of the method. In particular, we have adapted to our context suitable strategies, based on operators often used in static program analysis such as widening and convex-hull, for the automatic discovery of loop invariants of the imperative programs to be verified.

(ii) The implementation of our method into a prototype automatic tool, called VeriMAP, based on the CIL tool [115] and the MAP transformation system [108]. The current version of the VeriMAP tool can be used for verifying partial correctness properties of C programs that manipulate integers and arrays.

(iii) We have considered a significant fragment of the C Intermediate Language, and we have shown, through an extensive experimental evaluation on a large set of examples taken from different sources, that our approach, despite its generality, is also effective and efficient in practice.

The second contribution of this thesis is a *synthesis* framework based on Answer Set Programming (ASP), for the synthesis of concurrent programs satisfying some given behavioural and structural properties. Behavioural properties are specified by formulas of the Computational Tree Logic (CTL) and structural properties are specified by simple algebraic structures. The desired behavioural and structural properties are encoded as logic programs which are given as input to an ASP solver which, then, computes the answer sets of those programs. Every answer set encodes a concurrent program satisfying the given properties.

Possible developments and directions for future research

In the future we intend to develop, within the general framework based on CLP transformation, verification techniques that support other language features, including recursive function calls, concurrency, and more complex data types, such as pointers. These extensions require the enrichment our framework by considering some more theories, besides the theory of arrays, so that one can prove properties of programs manipulating dynamic data structures such as lists or heaps, by using for a set of suitable constraint replacement rules based on those theories. We think that an approach similar to the one we have used to handle array programs could be able to deal with different, more complex data structures. For some specific theories we could also apply the constraint replacement rule by exploiting the results obtained by external theorem provers or Satisfiability Modulo Theory solvers. As a further line of future research, we plan to explore the application of our method to (universal) termination proofs of imperative programs.

For what concerns our implementation of the verification framework, we have that the current version of VeriMAP is limited to deal with partial correctness properties of a subset of the C language. Moreover, the user is only allowed to configure the transformation strategies by selecting different submodules for unfolding, generalization, constraint solving, and replacement rules. Future work will be devoted to extend the VeriMAP tool towards a more flexible tool that enables the user to configure other parameters, such as: (i) the programming language and its semantics, (ii) the properties and the proof rules for their verification (thus generalizing an idea proposed in [74]), and (iii) the theory of the data types. Since program transformation essentially is a compilation technique, we believe that its use, together with a careful design of the parameters mentioned at Points (i)–(iii), can be a key factor in the automated generation of scalable program verifiers.

For what concerns our implementation of the synthesis framework, we observed that, in practice, it works for synthesizing k-process concurrent programs with a limited number k of processes because the grounding phase needed to compute the answer sets, requires very large memory space for large values of k. As a future work we plan to explore various techniques for reducing both the search space of the synthesis procedure and the impact of the grounding phase on the memory requirements.

Bibliography

[1] P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Regular model checking without transducers (On efficient verification of parameterized systems). In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '07*, Lecture Notes in Computer Science 4424, pages 721–736. Springer-Verlag, 2007.

[2] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of java bytecode using analysis and transformation of logic programs. In *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, pages 124–139. Springer, 2007.

[3] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *Proceedings of the 24th International Conference on Computer Aided Verification, CAV '12*, Lecture Notes in Computer Science 7358, pages 679–685. Springer, 2012.

[4] E. De Angelis, A. Pettorossi, and M. Proietti. Synthesizing concurrent programs using answer set programming. *Fundamenta Informaticae*, volume 120, number 3-4, pages 205–229, 2012.

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

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

[7] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20:51–115, January 1998.

[8] P. C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems*, 23:187–242, 2001.

[9] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[10] C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

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

[12] D. Beyer. Second competition on software verification (Summary of SV-COMP 2013). In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '13*, Lecture Notes in Computer Science 7795, pages 594–609. Springer, 2013.

[13] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '07*, Lecture Notes in Computer Science 4349, pages 378–394. Springer, 2007.

[14] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42. Springer-Verlag New York, Inc., 1987.

[15] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and assertions. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming, CP '95*, Lecture Notes in Computer Science 976, pages 589–623. Springer, 1995.

[16] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories, SMT-COMP '12*, volume 20 of EPiC Series, pages 3–11. EasyChair, 2013.

[17] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *Proceedings of the 20th International Symposium on Static Analysis, SAS '13*, Lecture Notes in Computer Science 7935, pages 105–125. Springer, 2013.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, Lecture Notes in Computer Science 2566, pages 85–108. Springer, 2002.

[19] P. Bonatti, F. Calimeri, N. Leone, and F. Ricca. Answer Set Programming. In *A 25-Year Perspective on Logic Programming*, Lecture Notes in Computer Science 6125. Springer, 2010.

[20] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI '06*, Lecture Notes in Computer Science 3855, pages 427–442. Springer, 2006.

[21] S. Brass and J. Dix. Characterizations of the Stable Semantics by Partial Evaluation. In *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science 928, pages 85–98. Springer, 1995.

[22] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '09*, pages 243–262. ACM, 2009.

[23] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(1):115–134, 1991.

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

[25] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer, Berlin, 1982.

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

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

[28] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of Logic in Computer Science, LICS '89*, pages 353–362. IEEE Computer Society, 1989.

[29] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification, CAV '00*, Lecture Notes in Computer Science 1855, pages 154–169. Springer, 2000.

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

[31] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL '79*, pages 269–282. ACM Press, 1979.

[32] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM Symposium on Principles of programming languages, POPL '11*, pages 105–118. ACM, 2011.

[33] P. Cousot, R. Ganty, and J-F. Raskin. Fixpoint-Guided Abstraction Refinements. In *Proceedings of the 14th International Symposium on Static Analysis, SAS '07*, Lecture Notes in Computer Science 4634, pages 333–348. Springer, 2007.

[34] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages, POPL '78*, pages 84–96. ACM, 1978.

[35] B. Cui and D. S. Warren. A system for tabled constraint logic programming. In *Proceedings of the 1st International Conference on Computational*

Logic, CL '00, Lecture Notes in Artificial Intelligence 1861, pages 478–492. Springer-Verlag, 2000.

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

[37] E. De Angelis. Software Model Checking by Program Specialization. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 12*, volume 17 of *LIPIcs*, pages 439–444, 2012.

[38] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Software Model Checking by Program Specialization. In *Proceedings of the 9th Italian Convention on Computational Logic, CILC '12*, pages 89–103. CEUR-WS, 2012.

[39] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with constrained generalization for software model checking. In *Proceedings of the 22nd International Symposium Logic-Based Program Synthesis and Transformation, LOPSTR '12*, Lecture Notes in Computer Science 7844, pages 51–70. Springer, 2013.

[40] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of imperative programs by constraint logic program transformation. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, SAIRP '13)*, Electronic Proceedings in Theoretical Computer Science, volume 129, pages 186–210, 2013.

[41] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of imperative programs through transformation of constraint logic programs. In *Proceedings of the 1st International Workshop on Verification and Program Transformation, VPT 13*, volume 16 of EPiC Series, pages 30–41. EasyChair, 2013.

[42] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13*, pages 43–52, New York, NY, USA, 2013. ACM.

[43] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. Technical Report 11, IASI-CNR, Roma, Italy, 2013.

[44] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Array Programs by Transforming Verification Conditions. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '14*, Lecture Notes in Computer Science 8318, pages 182–202. Springer, 2014.

[45] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Proceedings of the 15th 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '14*, Lecture Notes in Computer Science 8413, pages 568–574. Springer, 2014. To appear.

[46] E. De Angelis, A. Pettorossi, and M. Proietti. Using answer set programming solvers to synthesize concurrent programs. Technical Report 19, IASI-CNR, Roma, Italy, 2012.

[47] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, pages 337–340. Springer, 2008.

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

[49] G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.

[50] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[51] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewod Cliffs, N.J., 1976.

[52] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems, ESOP '10*, Lecture Notes in Computer Science 6012, pages 246–266. Springer, 2010.

[53] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning* (*KR '08*), pages 422–432. AAAI Press, 2008.

[54] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22:364–418, 1997.

[55] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[56] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

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

[58] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '00*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.

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

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

[61] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. *Fundamenta Informaticae*, 119(3-4):281–300, 2012.

[62] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

[63] C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1–3):253–270, 2004.

[64] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM Symposium on Principles of programming languages, POPL '02*, pages 191–202. ACM Press, 2002.

[65] J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93*, pages 88–98. ACM Press, 1993.

[66] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Conference of Artificial Intelligence* (*IJCAI '07*), pages 386–392. AAAI Press, 2007.

[67] M. Gebser, J. Lee, and Y. Lierler. Head-elementary-set-free logic programs. In *Proceedings of the 9th International Conference on Logic Programming and Nonmotonic Reasoning (LPNMR 2007)*, Lecture Notes in Computer Science 4483, pages 149–161. Springer, Berlin, 2007.

[68] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier, 2007.

[69] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[70] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.

[71] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, pages 345–377, 2006.

[72] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, POPL '05*, pages 338–350. ACM, 2005.

[73] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In *Proc. of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '12*, Lecture Notes in Computer Science 7214, pages 549–551. Springer, 2012.

[74] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416, 2012.

[75] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, pages 443–458. Springer, 2008.

[76] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09*, Lecture Notes in Computer Science 5643, pages 634–640. Springer, 2009.

[77] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the ACM Conference on Programming language design and implementation, PLDI '08*, pages 339–348, 2008.

[78] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11:157–185, 1997.

[79] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of pic programs through logic programming. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '06*, pages 103–179, 2006.

[80] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. In *Proceedings of the 9th Italian Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 3701, pages 280–294. Springer, Berlin, 2005.

[81] K. Hoder, N. Bjørner, and L. Mendonça de Moura. *μZ- An efficient engine for fixed points with constraints*. In *Proceedings of the 23rd International Conference on Computer Aided Verification, CAV '11*, Lecture Notes in Computer Science 6806, pages 457–462. Springer, 2011.

[82] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems. In *Proceedings of the 18th International Symposium on Formal Methods, FM '12*, Lecture Notes in Computer Science 7436, pages 247–251. Springer, 2012.

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

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

[85] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *Proceedings of the 24th International Conference on Computer Aided Verification, CAV '12*, Lecture Notes in Computer Science 7358, pages 758–766. Springer, 2012.

[86] J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.

[87] J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded Symbolic Execution for Program Verification. In *Proceedings of the 2nd International Conference on Runtime Verification, RV '11*, Lecture Notes in Computer Science 7186, pages 396–411. Springer, 2012.

[88] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP '09*, Lecture Notes in Computer Science 5732, pages 454–469. Springer, 2009.

[89] T. Janhunen and I. Niemelä. GnT – A Solver for Disjunctive Logic Programs. In *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science 2923, pages 331–335. Springer, 2004.

[90] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

[91] R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '06*, Lecture Notes in Computer Science 3920, pages 459–473. Springer, 2006.

[92] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV '07*, volume 4590 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2007.

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

[94] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[95] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, FASE '09*, Lecture Notes in Computer Science 5503, pages 470–485. Springer, 2009.

[96] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In *Applied Logic #16: Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.

[97] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[98] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, Rome, Italy, January 20-22, 2013*, Lecture Notes in Computer Science 7737, pages 169–188. Springer, 2013.

[99] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transaction on Computational Logic*, 7:499–562, 2006.

[100] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks, Release 3, Nov. 2000. Available from `http://www.ecs.soton.ac.uk/~mal`.

[101] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

[102] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.

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

[104] M. Leuschel and D. De Schreye. Constrained partial deduction. In *Proceedings of the 12th Workshop Logische Programmierung, WLP '97*, pages 116–126, Munich, Germany, September 1997.

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

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

[107] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[108] MAP. The MAP transformation system. `http://www.iasi.cnr.it/~proietti/system.html`. Also available via a WEB interface from `http://www.map.uniroma2.it/mapweb`.

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

[110] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[111] J. McCarthy. Towards a mathematical science of computation. In *Information Processing : Proceedings of IFIP 1962*, pages 21–28, Amsterdam, 1963. North Holland.

[112] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS '08*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2008.

[113] K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. MSR Technical Report 2013-6, Microsoft Report, 2013.

[114] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[115] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, Lecture Notes in Computer Science 2304, pages 209–265. Springer, 2002.

[116] I. Niemelä. Answer set programming without unstratified negation. In *Proceedings of the 24th International Conference on Logic Programming, ICLP '08*, volume 5366 of *Lecture Notes in Computer Science*, pages 88–92. Springer, 2008.

[117] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proceedings of the First International Conference on Computational Logic, CL '00*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.

[118] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR '02*, Lecture Notes in Computer Science 2664, pages 90–108. Springer, 2003.

[119] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, pages 246–261. Springer, 1998.

[120] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[121] A. Pettorossi and M. Proietti. Future directions in program transformation. In *Proceedings of the Workshop on Strategic Directions in Computing Research, MIT, Cambridge, MA, USA, June 14-15, 1996*, pages 99–102. ACM SIGPLAN Notices 32/1, January 1997.

[122] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989.

[123] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, pages 245–259. Springer, 2007.

[124] F. Ranzato, O. Rossi-Doria, and F. Tapparo. A forward-backward abstraction refinement algorithm. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '08*, Lecture Notes in Computer Science 4905, pages 248–262. Springer, 2008.

[125] T. W. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

[126] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74. IEEE Computer Society, 2002.

[127] A. Rybalchenko. Constraint solving for program verification: Theory and practice by example. In*Proceedings of the 22nd International Conference on Computer Aided Verification, CAV '10*, Lecture Notes in Computer Science 6174, pages 57–71. Springer, 2010.

[128] H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Symposium on Static Analysis, SAS '00*, Lecture Notes in Computer Science 1824, pages 377–396. Springer, 2000.

[129] J. S. Schlipf. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence*, 15:257–288, 1995.

[130] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *Proceeding of the 16th International Symposium on Static Analysis, SAS '2009*, Lecture Notes in Computer Science 5673, pages 3–18. Springer, 2009.

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

[132] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of the 1995 International Logic Programming Symposium, ILPS '95*, pages 465–479. MIT Press, 1995.

[133] T. Syrjänen. *Lparse 1.0 user's manual.* `http://www.tcs.hut.fi/Software/smodels/`, 2002.

[134] T. Syrjänen and I. Niemelä. The Smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmotonic Reasoning, LPNMR '01*, Lecture Notes in Computer Science 2173, pages 434–438. Springer, Berlin, 2001.

[135] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.

[136] M. Truszczyński. Logic programming for knowledge representation. In *Proceedings of the 23rd International Conference on Logic Programming, ICLP '07*, Lecture Notes in Computer Science 4670, pages 76–88. Springer, 2007.

[137] M. H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[138] VeriMAP. The VeriMAP software model checker. Available at `http://www.map.uniroma2.it/VeriMAP`.

[139] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI '04*, pages 131–144. ACM, 2004.

[140] M. Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, 1990.