A FRAMEWORK FOR PROGRAM REASONING BASED ON CONSTRAINT TRACES

ANDREW EDWARD SANTOSA

NATIONAL UNIVERSITY OF SINGAPORE

2008

A FRAMEWORK FOR PROGRAM REASONING BASED ON CONSTRAINT TRACES

ANDREW EDWARD SANTOSA
*(B.Eng., University of Electro-Communications,
M.Eng., University of Electro-Communications)*

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2008

Acknowledgments

I thank Prof. Joxan Jaffar for it has been a great privilege to enjoy his support throughout my doctoral study. I also thank Dr. Răzvan Voicu whose many ideas have inspired this thesis. I also thank Prof. Roland H. C. Yap, Prof. Abhik Roychoudury, Prof. Rafael Ramirez, Prof. Jinsong Dong, Prof. Gabriel Ciobanu, Prof. P. S. Thiagarajan, and Dr. Kenny Zhu, who have contributed to my education at the National University of Singapore. I thank Prof. Jinsong Dong also for the timed safety automaton example in Section 3.6.3. I thank Mihail Asavoae for the example in Section 3.5, and both Nguyen Huu Hai and Corneliu Popeea for discussions about parts of this work. I also thank Giridhar Pemmasani of the State University of New York at Stony Brook for his help on experimenting with XMC/RT, and Prof. Ranjit Jhala of the University of California at San Diego for his help on using Blast. I also highly appreciate brief but insightful interactions with Prof. David Dill of the Stanford University, and Prof. Andreas Podelski of the Max-Planck-Institut für Informatik. I am also indebted to the thesis examiners for their useful comments, including Prof. Cormac Flanagan of the University of California at Santa Cruz, and from the National University of Singapore: again Prof. Abhik Roychoudhury, Prof. Martin Sulzmann, Prof. Khoo Siau Cheng, and Prof. Martin Henz. Lastly, I am grateful for all my teachers and mentors in the past, especially Dr. Yasuro Kawata for training in research.

To Amelia

Contents

Summary

There have been many efforts in promoting the use of constraint logic programming (CLP) in program reasoning. There are two major approaches to program reasoning: path enumeration approach and syntax tree approach. Path enumeration is a search on state space of a program, and it can be accelerated by program analysis techniques, while syntax-tree (program verification)-based approach composes proofs of syntactic units, and is naturally compositional. We propose a CLP-based framework that accommodates both approaches.

Our framework is centered on a search-tree-based symbolic execution algorithm which performs generalization of execution state intermittently. Here our algorithm is engineered to function like an abstract interpreter for program analysis, with the main difference in that abstraction is applied intermittently, instead of at every analysis step. The advantages are that the abstract domain required to ensure convergence of the algorithm can be simplified, and that the cost of performing abstractions, now being intermittent, is reduced. Intermittent abstraction also enables compositional reasoning by viewing an abstraction point as a composition boundary.

The algorithm is optimized between the abstraction points using a novel *dynamic summarization* technique which summarizes a symbolic traversal subtree by generalizing its entry context such that more of newly encountered nodes in tree will be found to be subsumed and their correctness immediately concluded.

Our program reasoning framework can also employ optimization based on a novel notion of *relative safety*, which can significantly reduce the complexity of reasoning. We propose a framework which first lets the user specify *non-behavioral properties* such as symmetry, commutativity, or serializability as relative safety assertions, and prove the assertions automaticly. The proved assertions are then input to a traditional safety prover to obtain proof with reduced size. This allows us to handle more classes of symmetry than earlier approaches to symmetry reduction.

Our framework also handles verification of recursive data structures, which are specified recursively using CLP clauses. The verification technique is automatable. Our intermittent abstraction technique allows for simpler specification of recursive data structures, and solves the intermittence problem in data structure verification.

Our framework has the following formal underpinnings:

- Modeling of programs in CLP. Programs here include sequential and concurrent programs,

with or without underlying hardware constraints, and high-level specifications encompassing timed safety automata (TSA) and statecharts.

- Assertions to specify various correctness requirements. Their basic form is $G \models H$, where G and H are conjunctions of CLP atoms and constraints. We can use the assertions to express traditional safety (invariance) properties and relative safety (structural) properties of programs such as symmetry, commutativity, and serializability in concurrent programs. Since G or H may contain atoms of CLP predicates defining recursive data structures (linked lists, trees, etc.), the assertion can also be used to specify data structure properties.

- A proof method for general CLP programs. Our proof method can use an obligation, assumed to hold, to establish other obligations inductively. We call this process a *coinduction*.

We have developed a number of automated prover prototypes written purely in the CLP($\mathcal{R}$) programming language to demonstrate various aspects of our ideas, and we present the results of the experimental runs.

List of Programs

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problems

Having bugs in software is costly [186], and software failures have caused loss of life in safety-critical systems [76]. As the complexity of software systems increase, the quest for reliable software is becoming increasingly important. One of the techniques to improve systems reliability is *verification*, where logical reasoning is applied in order to prove *properties* of programs. This thesis draws a story about verification of systems, in particular, computer programs. Here we consider computer programs in a more general sense, encompassing concurrent programs, multiprocedural programs, timed programs whose behavior depends on the underlying hardware, as well as high-level behavioral models which are exemplified by *timed safety automata* (*TSA*) [99] and *statecharts* [88].

In program reasoning, the task is to prove whether a program satisfies a given property, that is, a statement about the program. There are two well-known classes of properties: *safety* and *liveness*. Informally, safety states that a bad thing does not occur, while liveness states that a good thing will occur. Formal definitions of both safety and liveness based on trace semantics have been given by Schneider who also shows that in trace semantics, properties can only be either safety or liveness [178]. This thesis focuses solely on safety properties. Here we define safety to be a subset of the state space of a program (that is, a subset where the "bad thing" does not occur). Some literature also categorizes statements about finite history of execution as safety, in particular the definition of safety using past temporal logic operators, e.g., in [19]. This is still consistent with our idea of safety since history can be recorded in computer memory, and hence can be viewed as a part of the state space.

There are two major approaches to safety verification in the literature. The first of these, which we call *path enumeration* approach, performs a search for error state ("bad thing") by computing all reachable states of the program starting from the initial state, or, in the reverse manner, performs a search for an initial state starting from the error state. All automatic reachability checkers (e.g., Murφ [49]) belong to this class. Path enumeration also includes temporal logic model checkers that proves the temporal logic formula $\Box\varphi$ with φ a proposition. Such formula states that the proposition φ holds in the initial state of the program and in all future states[1].

In path enumeration approach, each step of the search process is typically performed by *strongest postcondition* computation. The strongest postcondition $sp(t, \phi)$ is the state or *condition* (set of states) representing all possible next states after the execution of the statement t at state or condition ϕ. The search can also be done in a backward manner starting from the error state by computing at each step the strongest postcondition of the inverse of a statement (pre-image).

A major example of path enumeration approach is *model checking* [53]. It is based on state-space search given some properties to be proved. The state-space search is done on *concrete* program states. A concrete state is an assignment of every program variable to a constant in its domain, as opposed to *symbolic* state (condition), which is a constraint denoting a set of concrete states whose variable assignments satisfy the constraint. In model checking, the termination of the search is guaranteed due to the finiteness of of the domains. Model checking has been successful in hardware verification because here data domains can always be reduced to finite strings of binary digits. In contrast, software manipulates not only simple data such as numbers, but also arrays and pointer data structures. Representing these using binary digits ("bit-blasting") too easily results in a blowup of the size of the search tree. Therefore, for software verification, a symbolic traversal of the state space is more effective than concrete-state traversal. We note that strongest postcondition is applicable to either concrete or symbolic state-space traversal.

The path enumeration approach can be accelerated, and in case of infinite-state systems, its termination guaranteed, using *abstract interpretation* (*program analysis*) techniques [34]. This approach is based on providing abstract description of program states, where the concrete state space is mapped into an *abstract domain*. Reachability checking is then done on the abstract descriptions. Often such abstraction results in a finite number of possible abstract descriptions of program states (e.g., using an abstract domain that has a *finite lattice* structure), in which case

[1]There are two different interpretations of $\Box$ on whether it includes the present (initial) state or not. Here we assume it does.

the search is guaranteed to terminate. This technique is more efficient than normal reachability checking, but it is inherently incomplete due to the loss of accuracy incurred by the abstraction. We note that *shape analysis* [174] is an abstract-interpretation-based approach to data structure verification, but it suffers from inaccuracy [173, 86]. The challenge here is therefore the engineering of suitable abstract descriptions that make the traversal efficient, yet enable a proof.

More advanced abstract interpretation-based verifiers are based on *predicate abstraction* [80]. These incorporate an automated learning technique called *counterexample-guided abstraction refinement* (*CEGAR*) [30, 7, 97, 6] to try to compute a more appropriate abstract domain after every failure to prove a safety property. However, with predicate abstraction, it can be expensive to perform traversal on the abstract description where a single step of the search can be of exponential complexity to the number of predicates used in the abstract descriptions [9, 80].

In the area of path enumeration, other than abstract interpretation, data structures such as *binary decision diagram* (*BDD*) have been employed to make efficient both the propagation and the storage of the information collected during search, however, its applicability in software verification is limited. Another way to address the blowup problem is by enhancing the search technology. Explicit-state model checkers such as SPIN [101] employs *partial-order reduction* to reduce the search space. Some model checkers [96, 49, 28, 61] employ *symmetry reduction* for the same purpose. These reduction techniques do not lose precision, but their applicability is limited. Partial-order reduction mainly applies to communication protocols while symmetry reduction applies to mostly symmetric problems (e.g., distributed algorithms).

Another traditional branch of software verification technology is based on *program verification* [100]. This approach is a *syntax tree*-based, and it is employed in the verification of structured programs, that is, without arbitrary jump (**goto**) statements. Here, given a program fragment, a *precondition*, and a *postcondition*, we verify that any terminating execution of the program fragment in any state satisfying the precondition results in a state satisfying the postcondition. The correctness condition of a program fragment t is therefore specified as a *triple* $\{\phi\}\ t\ \{\psi\}$, where ϕ is the precondition, and ψ the postcondition. This technique can be used to verify programs where there is no guarantee of finiteness of data domain. The proof proceeds by applying several proof rules to obtain the desired conclusion. However, it is highly manual: some of the rules can be automated, but the rule to prove the correctness of loops especially requires the user to manually provide information.

Another challenge in program verification is symbolic computation of verification conditions.

One way of performing symbolic propagation is by *weakest precondition* computation, which is used in program verification tools such as ESC/Java [70] and Krakatoa [137]. A weakest precondition $wp(t,\psi)$ of a condition ψ and statement t is the weakest condition such that when t is executed from a state satisfying that condition, the resulting state either satisfies ψ or diverges (that is, t does not terminate normally). A triple $\{\phi\}\ t\ \{\psi\}$ holds if and only if $\phi \Rightarrow wp(t,\psi)$. The use of weakest precondition, however, is not a necessity. We can also employ strongest postcondition propagation in program verification since a triple $\{\phi\}\ t\ \{\psi\}$ holds also if and only if $sp(t,\phi) \Rightarrow \psi$.

We note that in contrast to path enumeration approach, the advantage of syntax tree approach is that it is compositional. For instance, the verification results of smaller fragments, which are specified as triples, can be used to establish the triple of their sequential composition.

Program verification is also amenable to data structure verification, such as using *separation logic* [166]. The reason is that any constraint, including those that are statements on state of data structures based on separation logic, is admissible as either pre- or postcondition. However, the automation of separation logic to date remains a challenge.

We summarize our discussion by listing the problems in program reasoning that we address in this thesis, namely

1. We address the efficiency of symbolic execution in three ways:

   (a) By a novel way of applying abstraction on symbolic states. As we have mentioned, one of the problems with abstract interpretation is engineering of suitable abstract domain that does not too quickly lose precision during symbolic traversal. Another problem is that one step of abstract traversal may be highly inefficient. Our objective is to simplify the abstraction used in the abstract traversal while maintaining precision, and also to increase the efficiency of each traversal step.

   (b) By a novel way of performing symbolic state-space exploration efficiently. New search algorithms are needed to expedite symbolic propagation. Note that symbolic propagation for verification is typically as complex as the verified program, which in turn is as complex as it can be (e.g., a programming solution to an NP-complete problem such as *subset sum problem*).

   (c) By a novel way of performing search-space reduction. As we have mentioned above, some reasoning systems employ symmetry and partial-order reduction. These techniques are applicable only to programs written in a specific syntax only. For programs where such properties are not obvious, the challenge is formal demonstration that they actually hold, so that they can be used for reducing the search space.

2. We also address the open problem of automatic verification of recursive data structures. We mention again that the main problem in shape analysis as with program analysis in general is information loss [27, 173, 86], while the main problem of separation logic is automation.

In addition, we want to reason on procedures or program fragments separately in order to simplify the whole proof by avoiding redundant proofs. It is therefore crucial to be able to perform compositional program reasoning in a similar sense to program verification.

1.2 Our Solution

In this thesis we propose a CLP-based approach toward solving the problems in program reasoning mentioned in the previous section. There have been many efforts in promoting the use of logic and constraint logic programming (CLP) for program reasoning. It is indeed natural to represent transition systems or deduction rules (e.g., to deduce the satisfaction of a temporal logic formula) as CLP clauses. For transition systems, the global transition relation is typically represented as a DNF formula, with each disjunct representing a state transition. It is straightforward to represent a state transition as a CLP clause. Similar to the symbolic execution of transition systems which visits program states, deductive proofs typically also contain a notion of a "state" of a proof containing formulas that have been deduced so far. CLP clauses can also be used to represent the transformation of such formulas.

Some of the existing CLP-based program reasoning approaches belong to the class of temporal logic model checkers, for example [46, 51, 65, 127, 149, 192]. Other than these, the approach of Gupta and Pontelli [84] can be considered as primarily a reachability checker. In fact, reachability checkers are straightforward to implement in (constraint) logic programming systems with resolution mechanism such as SLD.

Our program reasoning framework's main feature is symbolic traversal of state space by strongest postcondition propagation. Here we employ the correspondence of reduction in CLP execution to the computation of strongest postcondition. In general, symbolic strongest postcondition computation requires unbounded number of variables. For example, the resulting strongest postcondition of the statements $x := y+y$ followed by $y := 0$ is the condition $\langle \exists z : x = 2z \rangle \wedge y = 0$. In this way, a sequence of strongest postcondition computations may increase the number of existentially quantified variables in the symbolic state. CLP is suitable for implementing symbolic strongest postcondition computation since the variables are automatically maintained via an efficient projection mechanism.

The notion of strongest postcondition is also central in program verification since a triple $\{\phi\}\ t\ \{\psi\}$ holds if and only if $sp(t,\phi) \Rightarrow \psi$, as we have mentioned previously. This makes it possible to accommodate both the path enumeration approach and the syntax tree-based program verification approach in a single framework based on CLP. In this thesis we propose such framework.

We start our discussion with the formal foundations of our framework (Sections 1.2.1 and 1.2.2). We then expound on our main algorithm (Section 1.2.3), verification of data structures (Section 1.2.4), the proof and use of relative safety (Section 1.2.5), and we lastly discuss our implementation (Section 1.2.6).

1.2.1 Modeling Programs in CLP

We start by providing a methodology for modeling an extensive variety of programs in CLP. This include sequential and concurrent programs, multiprocedural programs, programs with hardware constraints on which they are run, programs with arrays and pointer data structures, even high-level specifications which include timed safety automata (TSA) [99] and statecharts [88].

We show an example modeling of a program in CLP in Programs 1.1 and 1.2, where Program 1.1 is a simple program with a **while** loop and Program 1.2 is its CLP model. In Program 1.1, $\langle l \rangle$ denotes a program point l. We assume that any program has an end point Ω. Here we map each statement in Program 1.1 into the corresponding CLP clause in Program 1.2. We also model a "condition of interest" as a CLP constraint fact. In Program 1.2, all states at the end point Ω is modeled by the constraint fact $p(\Omega, X, S, N)$.

High-level specifications such as timed safety automata and statecharts can be similarly translated into CLP programs.

1.2.2 Assertions and Proofs

After presenting the modeling of various kinds of programs in CLP, we proceed with their reasoning. The first thing that is required here is a way to formally specify the properties of the

```
Initially $x = s = 0$, $n \geq 0$.
⟨0⟩   while $(x < n)$ do
⟨1⟩      $s$ := $s + x$
⟨2⟩      $x$ := $x + 1$
      end do
```

Program 1.1: Sum

$$
\begin{array}{l}
p(\Omega, X, S, N).\\
p(0, X, S, N) \text{ :- } p(1, X, S, N), X < N.\\
p(0, X, S, N) \text{ :- } p(\Omega, X, S, N), X \geq N.\\
p(1, X, S, N) \text{ :- } p(2, X, S', N), S' = S + X.\\
p(2, X, S, N) \text{ :- } p(2, X, S, N), X' = X + 1.
\end{array}
$$

Program 1.2: Sum CLP Model

program. For this purpose we invent our own form of assertions to specify safety properties. Their basic form is $G \models H$, where G and H are *goals* (conjunctions of constraints and predicates interpreted by a CLP program). The intuitive meaning is that when G is true, so is H.

The simplest form of $G \models H$ that we use in this thesis is $p(\tilde{X}), \phi \models \psi$ where ϕ and ψ are purely conjunctions of constraints, while p is a predicate defined by the CLP model of a program. We call such assertions as *non-recursive assertions*. Such assertions represent what is known in the literature as *invariance* properties (cf. [144]). Any safety property is a form of invariance. In this thesis we also call invariance as *traditional safety*.

We also consider cases when ϕ or ψ contain predicates of CLP programs. We call such assertions as *recursive assertions*, and one of their use is in specifying traditional safety on recursive data structures. As a simple example, the assertion $p(H, Y) \models alist(H, Y)$ specifies that Y points to a head of an acyclic linked list on program heap H, where *alist* is defined by a CLP program.

Another form of recursive assertion is $p(\tilde{X}), \phi \models p(\tilde{Y}), \psi$ where p is defined by a CLP model of a program. We call such assertions as *relative* safety assertions. Relative safety specifies that a state satisfying ψ is reachable, if a state satisfying ϕ is. That is, it specifies relationships between states in the state space of the program. Relative safety can be uniquely used to assert structural properties of programs. We use relative safety to specify symmetry, commutativity, or serializability in a program. Relative safety allows us to represent and use larger class of symmetries than earlier approaches. Some mutual exclusion algorithms are a priority-based, destroying the symmetry among the concurrent processes. Here, a simple permutational symmetry (e.g., as handled by *scalarset* [107]) does not work. Nevertheless, some symmetry still holds, and we can specify and later prove this special kind of symmetry using relative safety assertions. We then employ the assertion for reduction in the verification run to prove the mutual exclusion property. We manage to prove the safety of two-process Szymanski's algorithm using symmetry reduction, which was not done previously.

We also devise a proof method to prove the assertions. The proof method is inductive, and it consists of a number of proof rules based on CLP resolution mechanism. More specifically, the proof of $G \models H$ proceeds by a number of *unfolding* steps of G to obtain a search tree with assertions $G_1 \models H, \ldots G_n \models H$ at the frontier. When $G_i \models H$ is unfolded from some ancestor $G_i' \models H$ and G_i is a special case of G_i', then we can apply inductive proof where we use $G_i' \models H$ as a hypothesis to prove $G_i \models H$. We call this inductive process as *coinduction*.

As a general CLP-based prover, the two main distinguishing characteristics of our proof method are the following two:

1. Some inductive proof methods are based on fitting in the allowable inductive proofs into an *induction schema* [118], which is usually syntax-based. Instead, we employ no induction schema. We detect the point of application of induction hypothesis using subsumption (e.g., of G_i by G_i' above). In other words, we discover the induction schema dynamically using indefinite steps of unfoldings. This approach is more powerful by the arbitrary number of unfolding steps, and more automatable by its algorithmic "search-based" nature.

2. We provide a goal generalization step which integrates very naturally into our framework. This adds into the completeness and efficiency of our proof method by allowing us to incorporate program analysis techniques. The same step is used to incorporate reductions such as symmetry reduction to improve efficiency.

1.2.3 Main Algorithm Based on Dynamic Summarization

The unfolding step of our proof method is based on reduction step in CLP execution, which, as we have mentioned, corresponds to strongest postcondition computation. This enables the combining of program analysis and verification in a single general algorithm based on our proof method.

When we are willing to compromise the completeness of the reasoning, we should be allowed to perform abstraction in the sense of program analysis to accelerate the reasoning. What is important here is the flexibility to apply abstraction intermittently. As mentioned above, it is often not easy to provide a suitable abstract domain so as to maintain accuracy. Program analysis loses information too quickly during the search process due to abstraction at each step of strongest postcondition computation. Applying abstraction only intermittently mitigates this problem. Also, this makes it not necessary to provide elaborate abstract domains to maintain accuracy.

Our algorithm can employ abstraction, such as predicate abstraction, and it can apply it intermittently. Here, our algorithm is engineered to function like an abstract interpreter, with the main difference in that abstraction is only applied at some program points. We repeat that the advantages here are that the abstract domain required to ensure convergence of the algorithm can be minimized, and that the cost of performing abstractions, now being intermittent, is reduced. Our work on intermittent abstraction has been reported in [113].

In this thesis we argue that the difference between abstract interpretation, program verification, and compositional (e.g., multiprocedural) program reasoning is simply the *location* at which abstraction is applied. In traditional abstract interpretation, abstraction is applied everywhere while in program verification the abstraction is typically done only at a point within each while loop whenever it is necessary to introduce *loop invariant*. A loop invariant is a condition that must be true at every iteration of a loop. Finally, in compositional program reasoning abstraction is performed at procedure call points or program fragment boundaries. In our flavor of compositional program reasoning, we prove assertion of the form $p(\tilde{X}'), q(\tilde{X}, \tilde{X}'), \phi \models \psi$, where $p(\tilde{X}')$ represents program predicate and $q(\tilde{X}, \tilde{X}')$ represents a predicate which is a CLP translation of a particular fragment of the program (e.g., a procedure). We first prove that $q(\tilde{X}, \tilde{X}')$ implies a transition relation $\rho(\tilde{X}, \tilde{X}')$ before proving $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), \phi \models \psi$ in place of the original assertion.

Between abstraction points, our algorithm performs exact (unabstracted) strongest postcondition propagation. We now discuss how we make this exact traversal efficient. We note that our algorithm constructs a proof tree with an assertion at each node. The proof of an assertion need not be pursued further when similar assertion has been established in the same tree. The efficiency of the verification process increases the more the similar assertions are. Here we design an optimization technique where we generalize proved assertions to increase the similarity of assertions encountered later in the proof. This technique is based on efficiently computing a precondition of paths in the proof tree. The computed precondition is more general than the context condition with which the analysis of the fragment is initiated. We call this technique as *dynamic summarization*. It has been reported in [112] as a central component of an overall technique to enhance the search efficiency for solving dynamic programming problems with ad-hoc constraints.

1.2.4 Verification of Recursive Data Structures

Our proof method is also engineered to handle verification of data structure properties represented as recursive assertions. For this purpose we define array as a basic data type in our CLP formalization, and we model the heap of the program as an array. A recursive pointer data structure such as lists or trees can then be specified as a CLP program which specifies the heap array.

Our algorithm can then be used for proving data structure properties. Although we only present an algorithm and not an automated implementation, our method is readily automatable in handling most data structure verification problems due to it being systematic, our reliance on CLP resolution, and the use of two principles: *array index principle* (*AIP*) and *separation principle* (*SEP*) to simplify proofs.

Some works mention “intermittence” (see e.g. [86]) as a limitation of shape analysis, and abstract interpretation methods in general. That is, due to the destructive nature of data-structure updates, invariants hold intermittently. Such examples are presented in [173], where the acyclicity of a tree is temporarily violated, and in [27], where an AVL tree becomes temporarily unbalanced. With intermittent abstraction, since we abstract only at specific (and small number of) program points (e.g., one point in each loop), and therefore we mostly compute exact information in the proof tree, we do not have to provide an elaborate set of predicates to avoid information loss. We demonstrate this using our proof of the AVL tree problem of [173] in Section 5.9.4.

In [173], it is also emphasized that shape analysis captures only the shape of the data structure, and not the contents, on which the correctness of the algorithm may depend. In our framework, it is straightforward to mix reasoning on data structure and its contents.

In the literature, data structure properties are often specified using an assertion language that allows recursive definitions [103, 147]. These formulations lead to using fold/unfold transformations to accomplish the proof [147]. Such transformations are used to achieve an inductive proof.

Existing fold/unfold transformations are only applicable in the case of recursive assertions that are “compatible” with the computation specified by the program. For instance, fold/unfold transformations would not prove a property of a linked list specified in a forward fashion, of a program that iterates backward through the list. In general, reasoning about programs annotated with recursive assertions remains an open problem because present methods are limited in applicability. We demonstrate an example where, using our proof method, we can use different recursion style in the recursive specification in order to solve the same verification problem.

The only CLP-based proof method that handles data structures that we are aware is the work of Hsiang and Srivas [103], which presents a framework for specifying Prolog data types and verifying it. The data structures here are limited to those definable using Prolog terms, and is not tailored for handling general pointer-based data structures in imperative languages. The framework allows users to write data structure specification which is then transformed into implementation. When the implementation is given by the user, the framework allows for the checking that it satisfies the specification. The verification process is the one presented in [102], which uses induction and manual variable marking to find the point of application of induction hypothesis. In contrast, we have developed an algorithm that is able to automatically discover, without manual intervention, a point in the proof where induction can be applied.

1.2.5 Relative Safety

Our proof method can be used to reason about symmetry, commutativity, and serializability of programs using the concept of relative safety. This allows us to specify and prove more classes of symmetry than can be handled by existing approaches. We then use the proved relative safety assertion for reducing the proof size of traditional safety assertions. Our work has been reported in [114] and partially in [111].

Existing approaches usually define symmetry on syntactic considerations. Semantically, symmetry is often defined as a transition-preserving equivalence [58, 29, 107, 60, 182], where an automorphism π, other than being a bijection on the reachable states, also satisfies that $(\tilde{x}, \tilde{x}')$ is a transition if and only if $(\pi(\tilde{x}), \pi(\tilde{x}'))$ is. Another notion of equivalence used is bisimilarity [55], replacing the second condition with bisimilarity on the state graph. These stronger equivalences allows for the handling of larger class of properties beyond safety such as CTL* properties. However, stronger equivalence also means less freedom in handling symmetries on the *collecting semantics* (set of reachable states), which we exploit further for proving safety properties. Because we handle symmetries on collecting semantics only, we obtain more flexibility in specifying various kinds of symmetries and employing them in state-space reduction, including symmetry in many problems that would not be considered symmetric by previous methods. We have mentioned above the Szymanski's mutual exclusion algorithm. We note that we can handle a wider range of symmetries than [55, 182]. More importantly, relative safety goes beyond symmetry because it also encompasses the property of commutativity and serializability, which is related to various techniques of reduction in literature [130, 155]. This work has been presented in [114]. The use of our proof method for symmetry reduction in TSA verification has also been reported in [111].

As mentioned Fribourg [75] (and also by Ramakrishna et al. [165]), when applied to the verification of finite-state systems, the goal of using CLP is to have a system written in a high-level language with declarative and flexible facilities while keeping good performance compared to specialized model checkers written in low-level code. This goal seems to have been partially achieved by systems like XMC [165], however, CLP-based systems still cannot compete with specialized model checkers. One of the reason, as mentioned by Fribourg, being *lack of integration with partial-order reduction techniques* [75]. Fribourg proposes the use of CLP resolution-based technique of *redundant derivation elimination*, but in this thesis we report an approach to reduction using commutativity and serializability.

1.2.6 Implementation

We have developed a number of automated prover prototypes written purely in CLP($\mathcal{R}$) [110] to demonstrate various aspects of our ideas. Our prototypes are used to automatically prove traditional and relative safety assertions. The proofs of traditional safety properties either employ relative safety properties (e.g., symmetry) for reduction or use dynamic summarization technique. Our implementations can be categorized as reachability checkers, but with advanced optimizations. We straightforwardly employ CLP resolution mechanism combined with meta-level features to symbolically manipulate constraints. In this thesis we also provide execution results of our prototypes.

1.3 Related Work

1.3.1 Related Work on CLP Prover for Program Reasoning

Related to our CLP proof method, is the class of work on reasoning about programs represented in CLP (see for example [75] for a non-exhaustive survey). Indeed, it is generally straightforward to represent program transitions as CLP clauses, and to use the CLP operational model to prove properties (as e.g., temporal logic) stated as CLP goals. Due to its capability for handling constraints, CLP has been notably used in verification of infinite-state systems [111, 45, 51, 65, 84, 127], although results for finite-state systems are also available [149, 165]. These however, are limited to certain representation of transition systems and cannot be used for proving general CLP programs. Moreover, these do not handle data structure verification.

We next review individual approaches.

We start with XMC [165], which is a model checker implemented on XSB logic programming system [175], taking advantage of SLG resolution mechanism implemented in XSB. The specification language of XMC is a CCS-like value-passing language, and properties are expressed using alternation-free mu-calculus. XMC/RT [51] is a version of XMC for the verification of timed safety automata given properties in timed mu-calculus. As with XMC, most temporal logic verification frameworks, in addition to representing the system to be verified in CLP, also represent the deduction rules of the temporal logic formula as CLP clauses. The verification is executed by a query on the deduction clauses.

Delzanno and Podelski [45, 46] present a CTL model checking method based on CLP. The CTL properties that can be proved are restricted to $\mathbf{AG}\phi$ and $\mathbf{AG}(\phi_1 \Rightarrow \mathbf{AF}\phi_2)$. The CLP representation of the system is transformed by adding rules representing the verification condition, and specialized algorithm is applied on the transformed representation to check the given property.

Nilsson and Lübcke also propose a method for CTL model checking using CLP [149]. The work treats semantically complete CTL, where it handles the **EX**, **EG**, and **EU** operators, which form an *adequate set* of CTL operators (see e.g., [105]). These are operators with a notion of existence, which can be easily formulated using CLP clauses. However, although Delzanno and Podelski succeeded in proving two-process bakery algorithm which is infinite-state, Nilsson and Lübcke's approach can only handle finite-state systems. The proof algorithm is based on transformation rules transforming a table containing *answers* and *goals*. The model checking is done locally (on-the-fly, picking one CLP clause at a time), yet uses symbolic model checking based on BDDs to perform CLP transformations.

Fioravanti et al. [65] propose another CTL verification approach using CLP *specialization*. Specialization is a program transformation technique whose objective is the adaptation of a program to the context of use. Note that CLP transformation may transform a program with a set of clauses onto a set of constrained facts representing the least model directly. Specialization of

Fioravanti et al. is done by adding a new rule to the CLP program describing the possible query. The program transformation is then used to infer that the head of the rule is in the *perfect model* semantics [4] of the CLP. The initial step of this approach is cross-producing the program to be verified with the CTL formula to derive the initial CLP clauses. The result is a CLP program with some resemblance to Nilsson and Lübcke's, but the approach is not restricted to only CTL operators with existential quantifier.

Finally we mention the work of Flanagan [66] which focuses on translating programs into CLP such that the least model of the CLP program is a relation of start state and end state of each block in the program. Given an error state, the CLP program is then transformed into another CLP program whose least model is all the possible initial states of every block in the program that leads to the error. The proof process proceeds by a query on the representation of the program's `main` block, constrained with the program's actual initial state. A refutation implies the reachability of the error state.

Satisfiability modulo theory (*SMT*) systems perform bounded (incomplete) automated verification based on SAT solving conjoined with theorem proving, yet the kind of theories that can be handled automatically and efficiently is limited [148]. The theory solving and the SAT solving in SMT systems are typically distinct. In CLP, they are tightly integrated, where theory (constraint) solving is performed at every step during the search. In this way, CLP avoids the problems introduced by multilevel satisfiability test typical to SMT solvers. We also note that CLP can be considered as a *lazy* approach to SMT where we there is no translation to boolean constraints necessary.

1.3.2 Related Work on TSA Verification Tools

Timed automata as defined by Alur and Dill in [2] are ω-automata [191] with continuous *clock variables* denoting elapsed time. In contrast to standard automaton, an ω-automaton accepts infinite words (known as ω-acceptance), as such ω-automata are used to represent the behavior of systems that runs forever. Accordingly, timed automata specify real-time systems that run forever. Timed *safety* automata (TSA) are timed automata without ω acceptance [99], therefore they are in essence transition systems. Reasoning of systems with continuous data domain as are TSA is natural to a CLP-based approach due to the required constraint solving. Prior to our work, TSA verification has been actively researched, and there are verification systems such as UPPAAL [13, 200], which is primarily a reachability checker, and symbolic model checkers for

TSA which include HyTech [98], Kronos [201] and RED [197]. In addition to these, there are also TSA verification tools based on CLP, including, which we detail next.

First, Gupta and Pontelli [84] presents a modeling of TSA in CLP. Although the work does not provide a systematic proof method, it demonstrates that in CLP-based system it is not necessary to use clock regions as in other timed automata verification systems [2, 200], since we can simply rely on the underlying constraint solving mechanism.

The work of Urbina [192, 193] is on verification of *hybrid automata* using CLP($\mathcal{R}$). Timed automata belong to a particular class of hybrid automata. They are called hybrid because the specification contains both discrete and continuous data values. A particular example of hybrid automata is timed automata. However, here the work treats automata with nonlinear physical properties. The framework allows for verifying Integrator Computation Tree Logic (ICTL) properties. The paper discusses proof methods for reachability, safety, duration properties, and ICTL properties. In our approach, we do not specify the constraints that can be handled. Our framework is also applicable to nonlinear constraints provided the solver is available.

A more systematic proof method for timed automata may require some form of tabling, as is presented with the XMC/RT model checker [51], which is based on the SLG resolution of XSB logic programming system [175]. It uses a generic constraint solver libraries written in C++ for solving linear arithmetic constraints over reals. XMC/RT represents TSA as a CLP program, and the properties are expressed using timed modal mu-calculus modeled in CLP. The work of Pemmasani et al. describes an improvement called XMC/dbm [156]. XMC/dbm includes an implementation for constraint solving using *Difference Bound Matrix* (*DBM*) [48], which is also employed in the UPPAAL model checker [200]. In contrast to our approach, the CLP tools we have mentioned here do not employ any form of reduction. Understandably, reduction is rather complex in general temporal logic verification.

1.3.3 Related Work on Symmetry in Verification

A well-known approach to symmetry-based reduction in model checking is based on *scalarsets* [107], which is implemented in the Murφ model checker. A scalarset is a qualifier of an index of a finite array. When an array has a scalarset index, exchanging the values of the array elements does not affect the truth value of the safety property being verified. That is, the array elements are *permutable* (hence scalarset approach handles *permutational* symmetry). Such array can be a list of program points, local variables of concurrent programs, or state of cache lines. In [107]

Ip and Dill specify syntactic properties that must be satisfied in the use of a scalarset.

Other model checker that employs symmetry is SMC [58, 85, 183, 184]. In SMC, permutation is restricted to process indices (not generally on array as with scalarsets), but in addition, some early detection of future symmetries during state space traversal is implemented. Here we note that symmetry induces *automorphism* mapping on the state reachability graph of a program. Two distinct states can be considered as symmetric when they can be mapped to each other by an automorphism. Emerson and Sistla describes how to identify automorphisms in CTL* formula [58, 59]. Although more variants of symmetries such as *rotational* symmetry and *reflective* symmetry were alluded to by Ip and Dill [107], the scalarset and SMC approaches both only handle permutational symmetry.

In some problems, not only array indices, but variable values must be permuted as well to obtain symmetry. For example, exchanging the value of some variable v from $v = 1$ to $v = 2$. This is called *permutation of variable-value pair* [182]. This permutation is also handled by TSA verification tools such as UPPAAL [96] and RED [196] in a limited way. RED handles symmetry is by assigning dynamic process ids to each concurrent process (an automaton in a system of automata) which are interchangeable (permutable) between the processes. When process 1 exchanges its process id with process 2, the variable $v = 1$ now points to process 2, since it now has id 1. RED, however, loses precision for problems with cyclic structure [198]. In contrast, our implementation does not lose precision due to symmetry.

Sistla and Godefroid attempt to handle systems whose state graphs are not fully symmetric in [182]. The approach transforms the state graph into a fully symmetric one, while keeping annotation for each transition that has no correspondence in the original state graph. The graph with full symmetry is then reduced by equating automorphic states. This work is the most general and can reduce the state graph of even totally asymmetric programs, however, the user has to statically specify transition priorities. In contrast, in our framework we prove the symmetry to be used in reduction.

Clarke et al. provides a way of inferring symmetry from the structure of the model, such as topology graph of concurrent processes [28] from the observation that structural symmetry introduces symmetry in the model to be verified. Still, however, the symmetries that can be handled by this approach is more limited than ours.

Manku et al. [133, 134] developed an algorithm to identify automorphisms in a hardware system specifications. Automorphisms are inferred from the rather simple structure of the circuits, where a function computed by a table can be represented as a graph. (In the case of software, we have no such convenience.) The algorithm succeeded in identifying rotational symmetry in a hardware version of the dining philosophers problem.

The work of Pandey and Bryant uses symmetry for the verification of transistor-level circuits [151]. Pandey and Bryant mentioned in brief a technique using symbolic simulation on transistor-level circuit to verify symmetry which is akin to our semantic proof of symmetry. However, they present no systematic method for this and focused more on inferring symmetry from circuit structure.

The work of Emerson et al. [55] also considers programs with non-obvious symmetries. The approach requires bisimilarity relationship between the original computation tree and the reduced computation tree. In our framework, we can do away with this requirement since we only deal with safety properties. The *virtual symmetry* considered by Emerson et al. is actually parameterized on a given automorphism group. Since automorphism group on state graph can be arbitrarily given, theoretically it can handle any system, either symmetric or asymmetric. It seems that here the problem of identifying symmetry itself is not given sufficient attention.

The work of Tang et al. [190] is on using symmetry for unbounded SAT-based model checker. The work mainly proposes an algorithm and makes no attempt at enlarging the set of symmetries that can be treated.

We repeat that the main difference between our work and these is that we propose a verification methodology where we prove that symmetry holds of a program. This is more powerful than imposing syntactic constraints to problems in order to apply symmetry reduction. Also since our proof method only verifies safety properties, we can identify more symmetries than is allowed in temporal logic verification-based setting.

1.3.4 Related Work on Reduction

Lipton presents an approach to group together some statements pertaining to one process in a concurrent program as single transition [130]. This is allowed when the interleavings of the statements with other processes are not necessary for verification. Since then, reduction techniques have been used for atomicity analysis [67, 68] and for improving the efficiencies of model checkers, known as *partial-order reduction* [53, 154, 155]. Both line of work are related to ours: The former concerns the proving of commutativity and serializability assertions, and the latter concerns the use of these assertions to expedite reasoning.

Ibarra et al. have identified that *commutativity checking* is undecidable in general [106] (e.g., with infinite-state systems). Atomicity analysis are often based on conservative tests that either miss atomicity violations or generate false alarms [68, 71, 164]. The work of Flanagan [67] is based on examining all interleavings and checking that the end result is the same state as executed by a serial execution. This is similar in essence to the approach we take in proving commutativity or serializability assertions.

Partial order reduction is a technique to reduce the search space in model checking. At each visited state, the model checker computes a subset of the enabled transitions at that state. Traversing only the subset preserves some (most commonly LTL$_{-X}$) properties. Partial-order reduction is based on the observation that concurrently executed transitions are often commutative because they are *independent*, e.g., do not access the same shared variable. Traditional implementations of partial-order reduction such as in model checker SPIN [101] is often based on statically defined dependencies of transitions, and the result is often too conservative. Flanagan and Godefroid proposed an algorithm for *dynamic* partial-order reduction algorithm [69], which can analyze dependencies more precisely. Dwyer et al. apply partial-order reduction upon detection of *thread locality* of heap data in concurrent Java programs by both static and dynamic means [52]. Here, a memory region is thread-local, if at any one time during execution, it is reachable from at most one thread only. Our technique of using commutativity and serializability properties for reduction can potentially be extended to any reduction that preserves the correctness of reachability check, including those that have been mentioned here.

We note that there has been an effort to combine static partial-order and symmetry reduction by Emerson et al. [56]. This is possible when the automorphism is bisimulation preserving, which is stronger than *stuttering equivalence* required by partial-order reduction [53]. Therefore, symmetry reduction can be augmented on top of a partial-order reduction model checking, and when some additional conditions are satisfied, preserves CTL^*_{-X} properties. In our framework, commutativity, serializability, and symmetry all belong to the class of relative safety properties. We can straightforwardly employ any combination of relative safety properties for search space reduction in verifying traditional safety properties.

1.3.5 Related Work on Compositional Program Reasoning

The compositional reasoning that we treat in this thesis is the independent reasoning of program fragments (e.g., procedures) which results are then used to reason about the whole program. A classic in this area is the work of Sharir and Pnueli [179] which consists of two approaches to interprocedural dataflow analysis. The first approach is called the *functional* approach, where the purpose is to establish input-output relation of each procedure. We then interpret a procedure call as an operation whose effect on program state can be computed using the relations. The second approach is orthogonal to the first. It is called the *call-string* approach. A call string is a sequence of procedure calls which reflects the status of the call stack. When a procedure is called with the same call stack, it is considered called with the same state. The call string is an abstraction of the program state, and therefore this approach is an approximation, but efficient in certain cases. Our compositional program reasoning technique is related to the first approach since we prove an assertion which states the input-output relation of a procedure. In the process, however, is optimized using dynamic summarization.

Although, as we have discussed, abstract interpretation is not naturally compositional, there is a work on compositionality for abstract interpretation which is done by Ball et al. [8]. The approach considers a second set of variables (called "symbolic constants"), in order to describe the input-output behavior of a procedure, in the language of predicate abstraction. As a comparison, our approach can also be tailored to utilize predicate abstraction to summarize a procedure by assertion. In addition, we use our novel dynamic summarization for optimization in proving assertions.

1.3.6 Related Work on Data Structure Verification

As we have mentioned above, there are two distinct approaches in the general area of reasoning about programs and data structures. One approach is based on logic, where new logical constructs are introduced and then integrated into a program verification-like proof system. Within this class, a recent prominent work is *separation logic* [166], whose outstanding feature consists of introducing logical connectives that describe non-sharing properties of data structures. However, as a program verification-based based calculus, it does not readily lend itself to automation. Moreover, separation logic does not explicitly support recursive assertions. Although the work of Guo et al. incorporates separation logic into shape analysis-like framework with arbitrary recursive predicates [83], but it is still not clear how to handle scalar values in their proof method.

Shape analysis (surveyed in [174]) is another class of solutions to the data structures reasoning based on abstract interpretation. Here the focus is on the accuracy/efficiency trade off involving the abstract domain, constructed from predicates that define the "shape" of the data structure, and the fixpoint iteration algorithm.

In general, shape analysis is global, in the sense that its predicates specifies the whole heap. It is therefore not easy to construct a modular, interprocedural shape analysis framework [50] because during an update of only one cell in the heap, the “shape” of the structure, which in fact determines the reachability relations of all variables, has to be recomputed. There have been attempts to introduce local reasoning into shape analysis by combining it with separation logic [168, 169, 83]. Separation logic mentioned above in contrast supports local reasoning well by means of the *frame rule*. For comparison, our recursive assertions are also global since it specifies the whole heap. However, the problem is mitigated by intermittent abstraction which supports compositional reasoning.

To address the intermittence problem in shape analysis (mentioned in Section 1.2.4), Chong and Rugina define an abstract domain consisting of a graph that specifies the reachability of the heap regions from the variables in the stack [27]. In this domain, the heap regions are assumed to be dynamic, for the purpose of handling destructive updates. Again here we mention that our intermittent abstraction solves the intermittence problem more straightforwardly.

Other approaches not yet mentioned include the approaches based on *graph types* [121, 145], which is based on program verification. PALE verifier [145] can be efficiently run when loop invariant is given. Intermittence problem still exists here, in which PALE allows the user to specify exceptions to invariants at certain program points, where they are temporarily violated. Reasoning about data fields are also allowed by some extension of PALE. PALE can handle only acyclic structures, or cyclic structure which are cyclic not by following the same field. In contrast, our approach is general.

McPeak and Necula presents an algorithm for specification and verification of data structure using equality axioms [140]. It has a better support for scalar values as compared to shape analysis. In this framework, however, temporary invariant breakage is still a problem. Dams and Namjoshi [40] propose shape analysis using predicate abstraction which is based on a set of basic recursive predicates stating reachability, sharing and cyclicity which are then used to define a set of derived predicates. A set of weakest precondition transformations of these predicates are defined. Our approach is more general by allowing user-defined predicates. Lahiri and Qadeer [124] propose the concept of *well-founded linked list* which is a (cyclic or acyclic) linked list whose “end” is signaled by a marker called “head.” This work considers only lists and does not explicitly consider separation. Hendren et al. propose *Abstract Description of Data Struc-*

tures (*ADDS*) [95] as another abstract interpretation based approach, whose abstract domain is the path matrix, which consists of the set of relations between pointers in the program and allows maintaining alias information which is then used for compiler optimization. Our approach to data structure verification can also be used to prove non-aliasing.

Finally, Jeannet et al. [116] propose an interprocedural shape analysis, based on representing each procedure as a structure on input and output predicates. However, their variant of shape analysis is storeless: there is no way to identify individuals in an input abstract structure with their corresponding individuals in the output abstract structure. In comparison, our approach in addition to being compositional, can also prove that an output heap is a modification of the input heap.

1.4 Structure of the Thesis

In Chapter 2 we start by providing an introduction to CLP with the domain of integers, terms, and arrays over integers. More domain, e.g., real and finite domain will be assumed in later chapters but left undefined. We build our exposition pedantically starting from the construction of predicate logic. In Chapter 3 we discuss how we model various programs and high-level specifications as CLP programs. In Chapter 4 we define our assertions, which can be used to specify traditional safety (invariance) properties, relative safety properties and properties of recursive data structures. We also discuss how it may represent some kind of liveness and equivalence. In Chapter 5, we present our proof method, whose core is a number of proof rules. We prove the soundness of our proof rules, and we exemplify the use of our proof method in proving traditional safety, relative safety, and properties on recursive data structures. We also present a theoretical foundation for compositional program analysis and verification which is based on intermittent abstraction, and which is the basis of our basic algorithm. In Chapter 6 we present a number of simple algorithms based on our proof method and the dynamic summarization technique, which gives rise to a general efficient algorithm for compositional program analysis and verification. The algorithm that is presented here proves non-recursive assertion. In Chapter 7 we discuss the automation of recursive assertion proofs, including relative safety and data structure assertions. In Chapter 8 we present the techniques used in implementing our prototypes, and the experimental result of the prototypes. We conclude this thesis and provide some future work in Chapter 9.

Chapter 2

Background in Constraint Logic Programming

In this chapter we will develop a formal language to write sets of formulas which constitute *constraint logic programs* (*CLP programs*). In our language we can specify objects called *arrays*. We therefore start by explaining a simple theory of arrays to be used throughout this report, elaborate the syntax and semantics of our language, and the explains an execution mechanism to prove the (un)satisfiability of our constraint logic programs.

Knowledge of constraint logic programming is useful (see the paper of Jaffar et al. [108]), although this chapter is generally rather pedantic. Here we assume familiarity with first-order logic and set theory. For readers new to first-order logic and its decision procedure, I would recommend a textbook by by Davis et al. [43]. Introduction to set theory can be found in Chapter 1 of the same book.

2.1 A Theory of Arrays

We first define a theory of arrays which will be used throughout this report.

We begin by denoting the set of natural numbers as $\mathbb{N}$, defined as $\{1,2,\ldots\}$, and the set of integers as $\mathbb{Z}$, defined as $\{\ldots,-2,-1,0,1,2,\ldots\}$.

An *array* is a function which has a finite support $\{0,1,\ldots,n\}$. We denote the support of an array a, as well as any function, as $sup(a)$. Therefore we can speak of the *size* of an array a, which is $|sup(a)|$. An array maps an element of its support to an integer value. For an array a, we write as $a[i]$, where $i \in sup(a)$, the mapping of i by a. We denote the set of all arrays as $\mathcal{A}$.

$$\begin{array}{lll}
\textit{Term} & ::= & \textit{Function Symbol}(\textit{Argument}_1, \ldots, \textit{Argument}_n) \\
\textit{Argument} & ::= & \textit{Variable} || \textit{Term} \\
\textit{Atom} & ::= & \textit{Relation Symbol}(\textit{Argument}_1, \ldots, \textit{Argument}_n) \\
\textit{Formula} & ::= & \square || \textit{Atom} || \textit{Formula}_1 \Rightarrow \textit{Formula}_2 || \\
 & & \langle \forall \textit{Variable} : \textit{Formula} \rangle || \langle \exists \textit{Variable} : \textit{Formula} \rangle
\end{array}$$

Figure 2.1: Syntax of Formulas

An array a can be updated at position $i \in sup(a)$ with an integer value e, resulting in another array a'. We represent an array update using a function $aupd : \mathcal{A} \times \mathbb{Z} \times \mathbb{Z} \mapsto \mathcal{A}$, takes as its arguments an array a, position i and argument e, and it maps these arguments to a new array a' which satisfies the conditions:

- $sup(a') = sup(a)$,
- $a'[i] = e$, and
- for all $j \in sup(a)$, when $j \neq i$, $a'[j] = a[j]$.

Arrays a and b are equal, denoted as $a = b$, if and only if a and b have the same support S, and for any $i \in S$, $a[i] = b[i]$ holds.

2.2 Formulas

We next define a simple language of formulas, whose syntax is defined using BNF in Figure 2.1. We use the term *subformula* when referring to a formula which is a syntactic component of another formula.

The syntax of nonterminals are obvious from Figure 2.1. We have therefore defined what we mean by a *term*, an *atom* or a *formula*.

Notice that a term or an atom has the syntax $f(\mu_1, \ldots, \mu_n)$, where f is a function symbol (for a term) or relation symbol (for an atom), and each μ_i is its *argument*. We attach a left-to-right ordering of the arguments which allow us to speak about the first, second, third or any i-th argument ($i \in \mathbb{N}$) of a term or atom t, denoted as $\arg(t, i)$. An *arity* of a term or atom is the number of its arguments. Given a term or atom t, we denote its arity by $\delta(t)$. A term or an atom with arity 0 is called a *constant*. A term (or atom) with arity 1, 2 or 3 is called a *unary*, *binary* or *ternary* term (or atom).

We next explain the terminals *Function Symbol*, *Relation Symbol* and *Variable* in that order.

We have three kinds of function symbols: *integer* (*arithmetic*) function symbols, *array* function symbols and *functor* symbols.

Integer function symbols are the set of all Arabic representation of integer numbers used to construct constants, the usual $+$, $-$, $\times$ and $/$ arithmetic operator symbols, and the symbol *aref*. The constants are 0-ary, while the arithmetic operator symbols are binary. Instead of writing $+(\mu_1, \mu_2)$, we use the infix notation $\mu_1 + \mu_2$. The terms constructed using *aref* are called *array references* and they are always binary. We would usually write *aref*(μ_1, μ_2) as $\mu_1[\mu_2]$.

There is no constant in the set of array function symbols, and here we only have the symbol *aupd* used to construct *array update expressions*. An array update expression always has arity 3, and has the syntax *aupd*(μ_1, μ_2, μ_3). We often write an array update expression as the triple $\langle \mu_1, \mu_2, \mu_3 \rangle$.

Functor symbols include at least the constant [] and the binary symbol *cons*. We would write *cons*(μ_1, μ_2) as $[\mu_1|\mu_2]$. Similarly, we write
cons$(\mu_1,$ *cons*$(\mu_2, \ldots$ *cons*$(\mu_n, [])\ldots))$ as $[\mu_1, \ldots, \mu_n]$.

Relation Symbol in Figure 2.1 is a relation symbol distinct from other symbols. There are two kinds of relation symbols: *interpreted* relation symbols and *uninterpreted* relation symbols. Interpreted relation symbols include integer arithmetic relation symbols $=$, $\leq$ and $\geq$[1]. We overload $=$ to also represent array and functor equality relation. Similar to arithmetic operator symbols, instead of writing $=(\mu_1, \mu_2)$, we use the infix notation $\mu_1 = \mu_2$. Interpreted relations also include the binary relation *size*, to be explained later. Uninterpreted relation symbols include symbols distinct from the rest.

Definiton 2.1 (Constraint). A *constraint* in our language is a formula or subformula not containing uninterpreted relation symbols.

We write a variable *Variable* as a sequence of Latin alphabet with a capital first letter which can be subscripted with a sequence of Arabic numbers, called its *index*. For example, both X and Var_{23} conform to the syntax of a variable. The index of the corresponding variable for the second syntax is 23. We attach a *type* to every variable, which is either integer, array, functor or any.

According to Figure 2.1, formulas may constructed using the symbols $\Box$ and $\Rightarrow$. The symbol

[1]Note that here we do not include $<$ or $>$ since $a < b$ if and only if $a+1 \leq b$, while $a > b$ if and only if $a \geq b+1$.

$\Box$ is called *falsum*, and it denotes logical falsity. The symbol $\Rightarrow$ denotes logical implication, that is, when $\alpha \Rightarrow \beta$ holds, then when the *antecedent* α holds, the *consequent* β must also hold. We will formalize these interpretations later.

For formulas α and β, we adopt the following shorthands:

- We write $\neg\alpha$ for $\alpha \Rightarrow \Box$.
- We write $\alpha \Leftarrow \beta$ for $\beta \Rightarrow \alpha$.
- We write $\alpha \wedge \beta$ for $\neg(\alpha \Rightarrow \neg\beta)$.
- We write $\alpha \vee \beta$ for $(\neg\alpha) \Rightarrow \beta$.
- We write $\alpha \Leftrightarrow \beta$ for $(\alpha \Rightarrow \beta) \wedge (\alpha \Leftarrow \beta)$.

Our *vocabulary* $\mathcal{W}$ is the set of all function and relation symbols.

Although our definition is sufficient for now, later we will expand our set of formulas as we see fit, such as adding real numbers, finite domain values and their operations.

Figure 2.1 allows us to write, within a formula γ, a *quantified* subformula β either using a *universal quantification of* X, where $\beta = \langle \forall X : \alpha \rangle$ or an *existential quantification of* X, where $\beta = \langle \exists X : \alpha \rangle$. A variable X has a *free occurrence* if it occurs in γ not within a subformula α of a quantification of X. A variable X which occurs in γ has a *bound occurrence* otherwise. A variable with free occurrence in γ is a *free variable* of γ. The set of all free variables of γ is denoted $var(\gamma)$. In writing formulas, we do not allow quantification of a variable when it is already bound. An example of a wrong formula with quantification of an already bound variable X is $\langle \forall X : X \leq 10 \Rightarrow \langle \forall X : X \geq 0 \rangle \rangle$.

We call a *sentence* a formula without free occurrences of variables. $\langle \forall X : X \leq 10 \Rightarrow p(X) \rangle \wedge \langle \forall X : X \geq 11 \Rightarrow q(X) \rangle$ is an example of a sentence.

2.3 Semantics of Formulas

We introduce here a *universe of discourse*, also known as *domain* $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$ of function and interpreted relation symbols. When f is a function or interpreted relation symbol, we denote by f^I its *interpretation* in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$.

2.3.1 Semantics of Constants

For all integer constant c, $c^I \in \mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$. Here, c^I is the number whose representation using Arabic numerals is c. Note that the integer constant interpretations in the domain $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$ is exactly the set $\mathbb{Z}$ of integers.

We assume that all arrays are in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$, that is, $\mathcal{A} \subset \mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$, although throughout this thesis we do not consider their syntactic representation. As we will see later, we can nevertheless precisely specify an array in a formula.

Any constant functor symbol has an interpretation in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$, which is an element of the set $\mathcal{F}$ defined as the least solution of the equation

$$\mathcal{X} = \{f(\mu_1, \ldots, \mu_n) | n \geq 0, \mu_i \in \mathbb{Z} \cup \mathcal{A} \cup \mathcal{X}, \text{ for all } 1 \leq i \leq n\}.$$

$\mathbb{Z}$ (integer), $\mathcal{A}$ (arrays) and $\mathcal{F}$ (functors) are called *basic types*. They are pairwise disjoint.

2.3.2 Semantics of Non-Constant Function Symbols

For each non-constant interpreted function symbol f in $\mathcal{W}$, a function $f^I : \mathcal{D}^{\delta(f)}_{\mathbb{Z},\mathcal{A},\mathcal{F}} \mapsto \mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$ is in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$. For example, the symbol "$+$" has as its interpretation (that is, $+^I$) the arithmetic operator $+ \in \mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$. Note that it is important here to distinguish between "$+$" as a syntactic element and its interpretation $+$. Similarly with "$-$", "$\times$" and "$/$." The semantics of expressions containing arithmetic operators are thus definable accordingly:

$$\begin{array}{rcl}
(\mu_1 + \mu_2)^I & = & \mu_1^I + \mu_2^I, \\
(\mu_1 - \mu_2)^I & = & \mu_1^I - \mu_2^I, \\
(\mu_1 \times \mu_2)^I & = & \mu_1^I \times \mu_2^I, \\
(\mu_1 / \mu_2)^I & = & \mu_1^I / \mu_2^I.
\end{array}$$

We adopt a precedence rule among the operators such that $\times$ and $/$ has a higher order of precedence than $+$ and $-$. Between $\times$ and $/$, the one written in the formula at the left of the other has a higher precedence. The same rule applies between $+$ and $-$. All interpretations of the arithmetic symbols are partial functions since they can only have integers as their arguments.

We have the following interpretations of array reference and array update:

$$(\mu_1[\mu_2])^I \quad = \quad \mathit{aref}(\mu_1^I, \mu_2^I),$$

$$\langle \mu_1, \mu_2, \mu_3 \rangle^I \quad = \quad \mathit{aupd}(\mu_1^I, \mu_2^I, \mu_3^I).$$

The interpretation of a functor symbol belongs to the set $\mathcal{F}$. Note, however, that there is a difference between a functor symbol and its representation. We write the representation of a functor symbol with the form $h(\mu_1, \ldots, \mu_n)$, for $n \geq 0$. This representation defines a function in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$ which constructs an element of $\mathcal{F}$ when some μ_i is substituted for an element of $\mathbb{Z} \cup \mathcal{A} \cup \mathcal{F}$. However, the functor symbol itself can be different from h, and its arity can be less than n. For example, $h(1, \mu, 2)$ can be a representation of a functor symbol $f(\mu)$. Now, the meaning of $f(g)$ (that is, $f^I(g^I)$), assuming $g^I = g$ is the constant $h(1, g, 2)$, which belongs to $\mathcal{F}$. Or similarly, the 0-ary functor symbol c may have as its representation $h(0,1) \in \mathcal{F}$[2].

From now on the difference between a functor symbol f and its interpretation f^I should be clear, but to avoid confusion, we will always consider functor symbols with the same symbol and arity as their representations. Each functor symbol is therefore interpreted into itself, but with each of its arguments interpreted in its appropriate domain. That is, for any functor symbol f,

$$(f(\mu_1, \ldots, \mu_n))^I \quad = \quad f(\mu_1^I, \ldots, \mu_n^I).$$

2.3.3 Semantics of Relation Symbols

For each non-constant interpreted relation symbol r in $\mathcal{W}$, a function $r^I : \mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}^{\delta(r)} \mapsto \{0,1\}$ is in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$. Informally, the value 0 denotes a "falsity" of the given relation while 1 denotes its "truth."

[2]In this way we also justify the 0-aryness of the constants in $\mathcal{F}$ since they are interpretations of 0-ary functor symbols.

For integer arithmetic comparators,

$$(\mu_1 = \mu_2)^I = \begin{cases} 1 & \text{if } \mu_1^I = \mu_2^I \\ 0 & \text{otherwise.} \end{cases}$$

$$(\mu_1 \leq \mu_2)^I = \begin{cases} 1 & \text{if } \mu_1^I \leq \mu_2^I \\ 0 & \text{otherwise.} \end{cases}$$

$$(\mu_1 \geq \mu_2)^I = \begin{cases} 1 & \text{if } \mu_1^I \geq \mu_2^I \\ 0 & \text{otherwise.} \end{cases}$$

When $=$ is used to compare arrays, the interpretation is as follows:

$$(\mu_1 = \mu_2)^I = \begin{cases} 1 & \text{if } \mathit{sup}(\mu_1^I) = \mathit{sup}(\mu_2^I) \\ & \quad \text{and } \langle \forall i \in \mathit{sup}(\mu_1^I) : \mu_1^I(i) = \mu_2^I(i) \rangle \\ 0 & \text{otherwise.} \end{cases}$$

Another interpreted relation symbol on arrays is *size*, interpreted as follows:

$$\mathit{size}(\mu_1, \mu_2)^I = \begin{cases} 1 & \text{if } \mathit{sup}(\mu_1^I) = \{0, 1, \ldots, \mu_2^I - 1\} \\ 0 & \text{otherwise.} \end{cases}$$

We also define the operator $=$ to be a syntactic equality test between functor symbols as follows[3]:

$$(\mu_1 = \mu_2)^I = \begin{cases} 1 & \text{if } \mu_1 \text{ and } \mu_2 \text{ have the same symbol and arity, and} \\ & \text{for each } i\text{-th argument, } (\mathit{arg}(\mu_1, i) = \mathit{arg}(\mu_2, i))^I = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Notice that we have overloaded the equality operator for comparison between integers, between arrays and between functors. Comparison between expressions which are interpreted to different basic types has the value 0.

We do not provide a domain for uninterpreted relation symbols, and they are supposed to

[3]This is the equality that separates functors from *uninterpreted functions*. Uninterpreted functions are equal if they are syntactically equal, otherwise, we do not know. Functors are equal if they are syntactically equal, otherwise they are not.

be interpreted in any suitable domain. Note, however, because they are relation symbols, in any domain they should be given an interpretation that returns a value in $\{0,1\}$.

2.3.4 Semantics of Formulas

Atoms are formulas, and we have given their semantics in the previous section. In this section we give the semantics of other kinds of formulas. The semantics for $\Box$ and $\Rightarrow$ is given below:

$$\Box^I = 0$$

$$(\mu_1 \Rightarrow \mu_2)^I = \begin{cases} 0 & \text{if } \mu_1^I = 1 \text{ and } \mu_2^I = 0 \\ 1 & \text{otherwise.} \end{cases}$$

Here the only interpretation of $\Box$ is 0 which denotes logical falsity.

Formulas α and β are semantically equivalent if and only if $\alpha^I = \beta^I$. We can express the semantics equivalence of α and β using the operator $\Leftrightarrow$ as $\alpha \Leftrightarrow \beta$. This is because $(\alpha \Leftrightarrow \beta)^I = 1$ exactly when $\alpha^I = \beta^I$.

From the above semantics of $\Box$ and $\Rightarrow$, we can prove the following semantics equivalences:

$$(\neg(\neg\alpha)) \Leftrightarrow \alpha \quad \text{(negation law),}$$

$$\begin{aligned} &(\alpha \wedge \beta) \Leftrightarrow (\beta \wedge \alpha), \\ &(\alpha \vee \beta) \Leftrightarrow (\beta \vee \alpha), \\ &(\alpha \Leftrightarrow \beta) \Leftrightarrow (\beta \Leftrightarrow \alpha) \quad \text{(commutative laws),} \end{aligned}$$

$$\begin{aligned} &(\alpha \wedge (\beta \wedge \gamma)) \Leftrightarrow ((\alpha \wedge \beta) \wedge \gamma), \\ &(\alpha \vee (\beta \vee \gamma)) \Leftrightarrow ((\alpha \vee \beta) \vee \gamma) \quad \text{(associative laws).} \end{aligned}$$

From the commutative and associative laws, there is no precedence in a sequence of conjunction or disjunction of formulas α_1 to α_n. We are allowed to then write $\alpha_1 \wedge \ldots \wedge \alpha_n$ as

$$\bigwedge_{i=1}^{n} \alpha_i.$$

Similarly, we can write $\alpha_1 \vee \ldots \vee \alpha_n$ as

$$\bigvee_{i=1}^{n} \alpha_i.$$

Other semantics equivalence include distributive and De Morgan's laws:

$$\begin{aligned}
&\alpha \wedge (\beta \vee \gamma) \Leftrightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma),\\
&\alpha \vee (\beta \wedge \gamma) \Leftrightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma),\\
&\alpha \vee (\beta \Rightarrow \gamma) \Leftrightarrow (\alpha \vee \beta) \Rightarrow (\alpha \vee \gamma),\\
&\alpha \Rightarrow (\beta \wedge \gamma) \Leftrightarrow (\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \gamma),\\
&\alpha \Rightarrow (\beta \vee \gamma) \Leftrightarrow (\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma),\\
&\alpha \Rightarrow (\beta \Rightarrow \gamma) \Leftrightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma) \quad \text{(distributive laws)},
\end{aligned}$$

$$\begin{aligned}
&\neg(\alpha \wedge \beta) \Leftrightarrow (\neg\alpha) \vee (\neg\beta),\\
&\neg(\alpha \vee \beta) \Leftrightarrow (\neg\alpha) \wedge (\neg\beta) \quad \text{(De Morgan's laws)}.
\end{aligned}$$

So far we have not completely define the semantics of formulas because we have not given any semantics to variables. A formula with variables is only interpretable when each variable is quantified. In other words, we only provide interpretation for sentences. We define the interpretation of non-sentence formula γ to be given by the interpretation of the sentence $\langle \forall var(\gamma) : \gamma \rangle$.

We now explain the semantics of quantification operators. We denote as $\{X \mapsto e\}$ the substitution of a variable X with a value $e \in \mathbb{Z} \cup \mathcal{A} \cup \mathcal{F}$.

$$\langle \forall X : \mu \rangle^I = \begin{cases} (\bigwedge_{e \in \mathbb{Z}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type integer} \\ (\bigwedge_{e \in \mathcal{A}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type array} \\ (\bigwedge_{e \in \mathcal{F}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type functor} \\ (\bigwedge_{e \in \mathbb{Z} \cup \mathcal{A} \cup \mathcal{F}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type any} \end{cases}$$

$$\langle \exists X : \mu \rangle^I = \begin{cases} (\bigvee_{e \in \mathbb{Z}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type integer} \\ (\bigvee_{e \in \mathcal{A}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type array} \\ (\bigvee_{e \in \mathcal{F}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type functor} \\ (\bigvee_{e \in \mathbb{Z} \cup \mathcal{A} \cup \mathcal{F}} \mu[X \mapsto e])^I & \text{if } X \text{ has the type any} \end{cases}$$

We would usually write $\langle \forall X_1 : \ldots : \langle \forall X_n : \alpha \rangle \ldots \rangle$ simply as $\langle \forall X_1, \ldots, X_m : \alpha \rangle$, and similarly for existential quantification.

Note that the following semantics equivalences hold:

$$\langle \forall X, Y : \alpha \rangle \quad \Leftrightarrow \quad \langle \forall Y, X : \alpha \rangle$$

$$\langle \exists X, Y : \alpha \rangle \quad \Leftrightarrow \quad \langle \exists Y, X : \alpha \rangle$$

Since in the above two cases the ordering of the quantifications of X and Y are unimportant, we may introduce a set $\tilde{X}$ which encompasses both the variables X and Y, and write formulas such as:

$$\langle \forall \tilde{X} : \alpha \rangle, \text{ or } \langle \exists \tilde{X} : \alpha \rangle.$$

When $\tilde{X}$ encompasses all of the free variables in α we write $\langle \tilde{\forall} \alpha \rangle$ for $\langle \forall \tilde{X} : \alpha \rangle$, and similarly for existential quantification.

Although it includes no constant arrays symbols, our language is powerful enough to express all arrays in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$. For example, a particular array can be specified using the following formula:

$$\langle \forall A : \langle \forall B : size(A,2) \wedge B = \langle \langle A, 0, 10 \rangle, 1, 20 \rangle \rangle \rangle.$$

B is certainly a variable which evaluates to an array, say b where $|sup(b)| = 2$ and $b(0) = 10$ and $b(1) = 20$.

2.4 Constraint Logic Programs

2.4.1 Definite Clauses

We first introduce *definite clauses* (also called *Horn clauses*), which are subformulas of the form

$$\langle \forall \tilde{X} : p \Leftarrow (q_1 \wedge \ldots \wedge q_n) \rangle,$$

where p is an atom of uninterpreted relation symbol called the *head* of the clause. The subformula $q_1 \wedge \ldots \wedge q_n$ is called the *body* of the clause, and each q_i, $0 \leq i \leq n$ is an atom. Note that therefore here we do not allow *negative literals*, where some q_i is of the form $\neg(r)$, with r an atom. It is possible that a clause has no body at all (the case when $n = 0$). We usually group together interpreted relation symbols in the body to the left of uninterpreted relation symbols (this does not change the semantics of the clause due to commutativity of conjunctions).

$\tilde{X}$ encompasses all variables occurring in the subformula of the quantification. When no q_i is an uninterpreted relation symbol, we call the definite clause as a *constrained fact*, or simply a *fact*. Note that a clause without a body is also a fact.

A *constraint logic program* is a conjunction of definite clauses. We usually call constraint logic programs as *CLP* programs, where *CLP* stands for *Constraint Logic Programming*. We now provide two examples of CLP programs.

Example 2.1. The following programs define a predicate p where $p(n)$ is true for some n if and only if n is a nonnegative even number.

$$\begin{aligned}&\langle\forall X : p(X) \Leftarrow X = 0\rangle\wedge\\&\langle\forall X,Y : p(X) \Leftarrow (X = Y + 2 \wedge p(Y))\rangle.\end{aligned}$$

Example 2.2. The following is a CLP program adapted from an example in [43]. The program states that all men are mortal, and Socrates is a man.

$$\begin{aligned}&\langle\forall X : mortal(X) \Leftarrow man(X)\rangle\wedge\\&man(socrates).\end{aligned}$$

2.4.2 Simplified Syntax

Further, we would write a definite clause without the variable quantification, write the implication symbol as `:-` , write conjunction symbol $\wedge$ using comma (,), and we end each clause with a period. Assuming the precedence of conjunction over implication, we also remove the parentheses enclosing the body of each clause.

Example 2.3. Following is an example of the program in Example 2.1, rewritten using new notational conventions:

$$\begin{aligned}&p(X) \texttt{ :- } X = 0.\\&p(X) \texttt{ :- } X = Y + 2, p(Y).\end{aligned}$$

The constraints in this program are $X = 0$ and $X = Y + 2$ (see Definition 2.1).

Example 2.4. We may also write the program of Example 2.2 in this way:

$$\begin{aligned}&\mathit{mortal}(X) \texttt{ :- } \mathit{man}(X).\\&\mathit{man}(\mathit{socrates}).\end{aligned}$$

The example has no constraints. It contains one functor: *socrates*.

2.5 Information Processing with CLP

We can regard a CLP program as a kind of "database" which stores knowledge that are of concern to us. We therefore need a mechanism to infer a piece of information from this database. In this section we first expound on the questions that we can pose to a CLP program. We then explain a decision procedure to answer the question.

2.5.1 Logical Consequence

Note that so far we have not really provide an interpretation to uninterpreted relation symbols. Whenever an uninterpreted relation symbol occurs in a constraint logic program, we may extend I with an interpretation for it. Therefore, we may speak of an non-unique extension I' of I which makes a sentence (in particular a constraint logic program) Γ true, that is, $\Gamma^{I'} = 1$. Such interpretation is called a *model* of Γ.

Given a CLP program Γ we write

$$\Gamma \Vdash \gamma$$

if every model of Γ is also a model of γ. As in the following two examples, we will further assume that γ is always a sentence containing no universal quantifier.

Example 2.5. Given Γ the program in Example 2.3 (Page 32), we may write

$$\Gamma \Vdash p(4).$$

In this case, we want to conclude that $p(4)^{I'} = 1$ for all model I' of Γ. One such model I' would interpret p as follows:

$$(p(X))^{I'} = \begin{cases} 1 & \text{if } X^I \text{ is a positive even number,} \\ 0 & \text{otherwise.} \end{cases}$$

Example 2.6. Similarto the above, given Γ the program in Example 2.2 (Page 32), we may write

$$\Gamma \Vdash \mathit{mortal}(\mathit{socrates}).$$

Here, based on information in Γ, we want to conclude that Socrates is mortal.

2.5.2 Resolution

We need a procedure to prove logical consequences. The one implemented in CLP systems is called *resolution*.

It is known that $\Gamma \Vdash \gamma$ holds if and only if $\Gamma \wedge \neg\gamma$ is *unsatisfiable*, meaning that there is no extension I' of I such that $(\Gamma \wedge \neg\gamma)^{I'} = 1$.

Notice that $\neg\gamma$ is equivalent to $\square \Leftarrow \gamma$. Further, when γ is an existentially quantified formula $\langle \exists \tilde{X} : \alpha \rangle$, the formula $\square \Leftarrow \gamma$ is equivalent to $\langle \forall \tilde{X} : \square \Leftarrow \alpha \rangle$. This can be written as a special CLP program clause

$$\square \texttt{ :- } \alpha.$$

We call clauses of this form as a *goal clause*. We call as a *goal* the body of a goal clause or any other conjunction of atoms. We add this clause into our CLP program, then apply resolution to the modified program to test its unsatisfiability.

The basic step of resolution is a generation of *resolvent* of two definite clauses κ_1 and κ_2. Here, the body of κ_1 must contain an atom of the same uninterpreted relation symbol with the head of κ_2. Moreover, we assume that κ_1 and κ_2 do not share variables. Whenever they do, we rename the variables in κ_2 appropriately to avoid sharing. Note that κ_1 and κ_2 can be the same clause, in which case we treat them as two separate copies and rename the variables appropriately.

Important to resolution is the notion of *unification* of two atoms or terms. Given two atoms or terms α and β, a substitution μ on their variables such that $\alpha\mu = \beta\mu$ is called a *unifier* of α and β. If α and β have some unifiers, then there exists a *most general unifier* (*m.g.u.*) among them[4]. A unifier μ_1 is more general than μ_2 if there is a substitution σ of the variables in μ_1 such that $\mu_1\sigma = \mu_2$.

[4]This is known as the *unification theorem*.

Now suppose that

$$\begin{aligned} \kappa_1 \quad &\text{is} \quad A \texttt{ :- } B_1, \ldots, B_n. \\ \kappa_2 \quad &\text{is} \quad C \texttt{ :- } D_1, \ldots, D_m. \end{aligned}$$

and that some $B_i = p(X_1, \ldots, X_l)$, and $C = p(Y_1, \ldots, Y_l)$. Suppose that the most general unifier for B_i and C is μ. The resolvent of κ_1 and κ_2 by the matching of B_i and C, denoted $\mathit{resolv}_{B_i}(\kappa_1, \kappa_2)$ is the new clause

$$A \texttt{ :- } B_1, \ldots, (B_{i-1}, B_{i+1}, \ldots, B_n)\mu, \ldots, X_l = Y_l, D_1, \ldots, D_m.$$

When Γ is a CLP program which includes the special clause $\square \texttt{ :- } \alpha$, a sequence of clauses $\kappa_1, \kappa_2, \ldots, \kappa_n = \kappa$ is called a *resolution derivation of* κ if for each i, $1 \leq i \leq n$, either κ_i is a clause in Γ or a resolvent of κ_j and κ_k, where $j, k < i$. A resolution derivation of $\square$ from Γ is called a *resolution refutation of* Γ.

We have the following *resolution theorem*, which is immediate from J. A. Robinson's general resolution theorem.

Theorem 2.1 (Resolution Theorem). A CLP program Γ is unsatisfiable if and only if there is a resolution refutation of Γ.

Example 2.7. We now prove the logical consequence in Example 2.5 (Page 33), where our CLP program is as follows.

$$\begin{array}{ll} \square \texttt{ :- } p(4). & \kappa_1 \\ p(X) \texttt{ :- } X = 0. & \kappa_2 \\ p(X) \texttt{ :- } X = Y + 2, p(Y). & \kappa_3 \end{array}$$

We generate the following clauses using resolution, hence deriving a $\square$, which proves the unsatisfiability of the above CLP program.

$$\begin{array}{ll} \square \texttt{ :- } p(2). & \kappa_4 = \mathit{resolv}_{p(4)}(\kappa_1, \kappa_3) \\ \square \texttt{ :- } p(0). & \kappa_5 = \mathit{resolv}_{p(2)}(\kappa_4, \kappa_3) \\ \square & \kappa_6 = \mathit{resolv}_{p(0)}(\kappa_5, \kappa_2) \end{array}$$

Example 2.8. Now we are ready to prove that Socrates is mortal. Here we are trying to prove the unsatisfiability of the CLP program:

$$\begin{array}{ll} \Box \texttt{ :- } \mathit{mortal}(\mathit{socrates}). & \kappa_1 \\ \mathit{mortal}(X) \texttt{ :- } \mathit{man}(X). & \kappa_2 \\ \mathit{man}(\mathit{socrates}). & \kappa_3 \end{array}$$

Using resolution, we derive the following clauses from the above CLP program, to prove its unsatisfiability.

$$\begin{array}{ll} \Box \texttt{ :- } \mathit{man}(\mathit{socrates}). & \kappa_4 = \mathit{resolv}_{\mathit{mortal}(\mathit{socrates})}(\kappa_1, \kappa_2) \\ \Box & \kappa_5 = \mathit{resolv}_{\mathit{man}(\mathit{socrates})}(\kappa_4, \kappa_3) \end{array}$$

2.5.3 SLD Resolution

Notice that in the two examples in the previous section, in each step we always generate a resolvent of a clause of a goal clause with a clause of the original CLP program, resulting in another goal clause, until finally we obtain a goal clause of the form $\Box \texttt{ :- } \alpha$, where $\alpha^I = 1$, and since $\Box^I = 0$, this expression is equivalent to $\Box$. Therefore here our use of $\kappa_3 = \mathit{resolv}_A(\kappa_1, \kappa_2)$ is restricted to the case when κ_1 is a goal clause which contains the atom A in its body, κ_2 is a clause of the original program, and the resulting resolvent κ_3 is a goal clause. Given a certain CLP program, we say that κ_3 is a *reduct* of κ_1.

In computing the reduct of a goal clause, in general we need to apply some or all of the clauses of the original CLP program. To implement resolution as a sequential algorithm, we need to determine an ordering among program clauses which determine their order of application. Similarly, given a goal clause, we also need to determine which uninterpreted atom in its body is to be matched first. A well-known implementation called *SLD resolution* applies the program clauses in top to bottom in the program text, and selects the atom in the goal clause's body from left to right in the program text. SLD is an abbreviation of *Selected-literal Linear resolution for Definite clauses*. In our restricted case literals are simply atoms and "definite clauses" here means definite clauses.

Since there are a number of derivations which are possibly infinite, an implementation has to try each derivation one by one until it finds one which ends in $\Box$. This is accomplished in CLP systems by a backtracking mechanism which, when a derivation fails, returns to the deepest goal where it is still possible to select another clause from the program to generate a new resolvent. SLD resolution with backtracking can be efficiently implemented using a stack which stores goals.

We note that SLD resolution is not guaranteed to discover a refutation even though one exists, not even when the set of the provable formulas (that is, the γ in logical consequence $\Gamma \Vdash \gamma$, where Γ is a CLP program) is recursive.

In addition to implementing resolution, CLP systems also utilize constraint solving algorithms to simplify interpreted expressions at each derivation step.

Example 2.9. Now let us redo Example 2.7 (Page 35) using SLD resolution and constraint solving, showing only the goals.

$\Box$:- $p(4).$	κ_1
$\Box$:- $4 = 0.$	$\kappa_4 = \mathit{resolv}_{p(4)}(\kappa_1, \kappa_2)$
$\Box$:- $\Box$.	Simplify κ_4 with constraint solving: Proof fails.

Our proof fails at a first attempt since $(\Box \text{ :- } \Box)^I = 1 \neq \Box^I$. Fortunately, resolution engines are equipped with backtracking mechanism which can return to an earlier goal and re-try using different program clause. Now, we return to κ_1 and use κ_3 instead of κ_2 to generate a new goal.

$\Box$:- $p(4).$	κ_1
$\Box$:- $4 = Y + 2, p(Y).$	$\kappa_5 = \mathit{resolv}_{p(4)}(\kappa_1, \kappa_3)$
$\Box$:- $p(2).$	Simplify κ_5 with constraint solving.
$\Box$:- $2 = 0.$	$\kappa_6 = \mathit{resolv}_{p(2)}(\kappa_5, \kappa_2)$
$\Box$:- $\Box$.	Simplify κ_6 with constraint solving: Proof fails.

Again the proof fails. Fortunately, we have not exhausted all possibilities. We backtrack to κ_5 and try to generate a resolvent using program clause κ_3 instead of κ_2, resulting in the derivation

$\Box$:- $p(4)$.	κ_1
$\Box$:- $4 = Y+2, p(Y)$.	$\kappa_5 = \mathit{resolv}_{p(4)}(\kappa_1, \kappa_3)$
$\Box$:- $p(2)$.	Simplify κ_5 with constraint solving.
$\Box$:- $2 = Y+2, p(Y)$.	$\kappa_7 = \mathit{resolv}_{p(2)}(\kappa_5, \kappa_3)$
$\Box$:- $p(0)$.	Simplify κ_7 with constraint solving.
$\Box$:- $0 = 0$.	$\kappa_8 = \mathit{resolv}_{p(0)}(\kappa_7, \kappa_2)$
$\Box$.	Simplify κ_8 with constraint solving.

Since we have obtained the goal $\Box$, the unsatisfiability proof succeeds.

2.6 Least Model

We next explain the *least model semantics* of CLP programs. Suppose that I' is an extension of I (that is, $I \subseteq I'$) which interprets the uninterpreted relation symbols in Γ such that $\Gamma^{I'} = 1$. A *model* of the CLP program Γ is the set $I' - I$. The *least model* of Γ is the strongest such model, which always exists for any CLP program with no negative literal [109].

Let us now proceed more formally.

Definiton 2.2 (Ground Substitution). Given a formula α, the *ground substitution* σ of α denoted $\sigma(\alpha)$ or $\alpha\sigma$ substitutes each free variable in α with a constant in $\mathcal{D}_{\mathbb{Z},\mathcal{A},\mathcal{F}}$.

Definiton 2.3 (Immediate Consequence Operator). Given a program Γ, we define an *immediate consequence operator* T_Γ, which takes an interpretation J for uninterpreted relation symbols mentioned in Γ and produces another one, as follows:

$$\begin{array}{rcll}
T_\Gamma(J) & = & \{\sigma(A) & \mid \; A \text{ :- } c_1, \ldots, c_n \text{ is a fact in } \Gamma \\
& & & \quad \text{and for all } i, 1 \le i \le n, (\sigma(c_i))^I = 1\} \cup \\
& & \{\sigma(A) & \mid \; A \text{ :- } L_1, \ldots, L_m, c_1, \ldots, c_n \text{ is in } \Gamma \\
& & & \quad \text{and for all } i, 1 \le i \le m, m \ge 1, \text{ and } \sigma(L_i) \in J, \\
& & & \quad \text{and for all } j, 1 \le j \le n, (\sigma(c_j))^I = 1\}
\end{array}$$

Here, A and $L_1, \ldots, L_m$ are atoms, c_i for any i are constraints, and σ denotes a ground substitution.

We denote by $T_\Gamma \uparrow n$ the set $T_\Gamma^n(\emptyset)$, where $T_\Gamma \uparrow 0 = \emptyset$.

Definiton 2.4 (Least Model). Given a program Γ, the *least model* of Γ is the least solution of the equation $X = T_\Gamma(X)$. Further, we denote the least model of Γ as $lm(\Gamma)$.

Because of the monotonicity and continuity of the T_Γ operator when Γ does not contain negative literals, then given a CLP program Γ without negative literals, by Knaster-Tarski fixpoint theorem (see e.g., [105]), when ω is an infinite ordinal, then $T_\Gamma \uparrow \omega = lm(\Gamma)$. For some CLP programs, $T_\Gamma \uparrow k = lm(\Gamma)$ for some $k < \omega$.

2.7 Clark Completion

By defining CLP programs as logical formulas, we have actually provided their logical semantics. Here we provide an even stronger logical semantics of CLP programs called Clark completion. The logical semantics we have given so far has too many models, which is undesirable in verification since to establish a property of a program, as we will see later, we need to inspect all possible behavior of the program. With our current logical semantics (conjunction of implications), the CLP representation of a program to be verified would imply many other behavior which does not exist as a behavior of the original program. For example, a possible model I' of the CLP program in Example 2.1 (Page 32) is $p(X)^{I'} = 1$ for any X. However, this is not the *intended* model of the CLP program. Instead, what we have in mind is the model given in Example 2.5 (Page 33), which is indeed the least model.

Using Clark completion as the logical semantics would restrict the possible models to the least model only. The following discussions in this section can also be found in [109].

Definiton 2.5 (Completion). The logical semantics of n-ary predicate symbol p in the program Γ is the formula

$$\langle \forall X_1,\ldots,X_n,\tilde{Y} : p(X_1,\ldots,X_n) \Leftrightarrow B_1 \vee \ldots \vee B_m \rangle,$$

where $\tilde{Y}$ are variables in $B_1,\ldots,B_m$, and $X_1,\ldots,X_n$ are variables that do not appear in any clause. Except when they belong to $X_1,\ldots,X_n$, the variables of $\tilde{Y}$ in B_i is disjoint from the variables in B_j, whenever $i \neq j$, which can be achieved by appropriate renaming. Further, each B_i corresponds to a clause in Γ of the form

$$p(t_1, \ldots, t_n) \texttt{ :- } L_1, \ldots, L_k.$$

and B_i is

$$X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge L_1 \wedge \ldots \wedge L_k.$$

If there is no clause with head p, the completion of p is simply

$$\langle \forall X_1, \ldots, X_n : p(X_1, \ldots, X_n) \Leftrightarrow \Box \rangle.$$

The *Clark completion* Γ^* of a CLP program Γ is the conjunction of the above definitions of all uninterpreted relations in Γ.

A known result is that when Γ^* is the Clark completion of Γ, $lm(\Gamma^*) = lm(\Gamma)$. Most importantly, establish the relation $\Gamma^* \Vdash \langle \tilde{\exists} \gamma \rangle$ if and only if $lm(\Gamma^*) \Rightarrow \langle \tilde{\exists} \gamma \rangle$.

2.8 Further Readings

We note that the array functions [] and *aupd* introduced in Section 2.1 have appeared in the literature. They are are similar to the and *select* and *store* of Nelson and Oppen [146], or *read* and *write* of Jones et al. [117].

For further reading on the semantics of CLP programs, refer to Jaffar et al. [109]. For further reading on the basics of resolution and refutation, readers may refer to Davis et al. [43] as well as Boolos and Jeffrey [20]. Introduction to constraint solving algorithms can be found in the book by Marriott and Stuckey [138].

Chapter 3

Modeling Programs in CLP

To use CLP for program reasoning, we need to first model programs as CLP programs, such that some semantics correspondence exists between the original program and its CLP model. In this chapter we show how we model various programs and high-level specifications.

3.1 Sequential Programs

3.1.1 Usual Semantics

We first define a simple sequential programming language whose syntax is given in Figure 3.1. Note that particular to our language, we may (or may not) consider a sequence of assignments as just a single statement. In our language, we annotate each statement of the program with a unique positive integer *program point label* enclosed in angle brackets. It can be considered as an address of the statement relative to the start of the code segment of the program. We assume there is a special program point Ω at the end of each program.

Program 3.1 is an example of a program written in this syntax. Program 3.1 has x, s, n and a special variable l as *program variables*. The variable l is used to store the program point label of the next statement to be executed. Before the program text we usually provide a comment using "Initially" keyword, on the initial execution states of the program. For example, Program 3.1 starts with $x = s = 0$ and $n \geq 0$. We always assume that the initial value of l is the first program point of the program, which in Program 3.1 is 0.

Definiton 3.1 (States and Conditions). A *program state* (or simply state) is a substitution σ of each program variable in the corresponding domain. We often represent a state using a

```
         Prg  ::=  LabeledStmt_1 ... LabeledStmt_n

 LabeledStmt  ::=  ⟨Label⟩ Stmt ||
                   goto ⟨Label⟩

        Stmt  ::=  AssignSeq ||skip ||
                   if (BoolExpr) then Prg end if ||
                   if (BoolExpr) then Prg else Prg end if ||
                   while (BoolExpr) do Prg end do

   AssignSeq  ::=  Variable := Expr ||
                   Variable := Expr   AssignSeq
```

Figure 3.1: Simple Programming Language

```
Initially $x = s = 0$, $n \geq 0$.
⟨0⟩  while $(x < n)$ do
⟨1⟩      $s$ := $s + x$
⟨2⟩      $x$ := $x + 1$
     end do
```

Program 3.1: Sum (Repeat of Figure 1.1)

constraint that is true if and only if the substitution is applied to the variables in the constraint. A *condition* is a set of states, hence a set of substitutions. We represent a condition as a constraint that is true when any of the substitution in the set is applied to the variables.

Definiton 3.2 (Transitions). A *transition* is a relation between a *source* (*pre*) and a *target* (*post*) state. We represent it as a constraint on two sets of variables: program variables and their primed versions. We usually adopt a more general notion of transition which relates two conditions. A *transition relation* is a set (disjunction) of transitions, denoted $\rho(\tilde{x}, \tilde{x}')$, where $\tilde{x}$ is the sequence of program variables and $\tilde{x}'$ is the sequence of their primed versions.

We denote the transitive closure of transition relation ρ as ρ^*.

A transition represents a change of program variable values from a source (pre) to a target (post) condition. In this way we may define the notion of “computation” of a program by a sequence of transitions, which starts from the initial condition.

Definiton 3.3 (Reachable State). A state s_2 is *reachable* from a state s_1 when $s_2 \Rightarrow \langle \exists \tilde{x}' : (s_1[\tilde{x} \mapsto \tilde{x}'] \wedge \rho^n(\tilde{x}', \tilde{x})) \rangle$ for some $n \geq 0$. In particular, s is a *reachable state* of a program if and only if $s \Rightarrow \langle \exists \tilde{x}' : (\Theta[\tilde{x} \mapsto \tilde{x}'] \wedge \rho^n(\tilde{x}', \tilde{x})) \rangle$ for some $n \geq 0$, where Θ is the initial condition of the program.

So far we have actually provided the building blocks of a deterministic *discrete dynamic system* [33], with a set of states (all possible substitutions on program variables), transition relation, entry condition, and exit condition ($l = \Omega$). Hence, following [33], we may characterize the set of reachable states of the program as strongest interpretation of α such that

$$\alpha = \Theta \vee \langle \exists \tilde{x}' : \alpha[\tilde{x} \mapsto \tilde{x}'] \wedge \rho(\tilde{x}', \tilde{x}) \rangle.$$

Here, Θ is the initial condition of the program. We denote such strongest interpretation as $\mathit{Conv}(\rho, \Theta)$. Therefore we can also say that a state s is reachable if and only if $s \Rightarrow \mathit{Conv}(\rho, \Theta)$.

We may also define a “condition of interest” Ξ of a program. We may then want to analyze the set from where the condition of interest may be reached. The condition of interest can be an exit condition such as in [33], but this can be generalized to any condition. Following the result of [33], we characterize the set of “ancestor” states of a condition of interest as the strongest interpretation of α such that

$$\alpha = \Xi \vee \langle \exists \tilde{x}' : \alpha[\tilde{x} \mapsto \tilde{x}'] \wedge \rho(\tilde{x}, \tilde{x}') \rangle.$$

We denote such strongest interpretation as $\mathit{Conv}(\rho^{-1}, \Xi)$. A state s *reaches* Ξ if and only if $s \Rightarrow \mathit{Conv}(\rho^{-1}, \Xi)$.

Example 3.1. For example, a transition representing all possible state changes between program points 0 and 1 in Program 3.1 can be represented as the constraint $l = 0 \wedge x < n \wedge l' = 1 \wedge x' = x \wedge s' = s \wedge n' = n$. In this transition, the values of the program variables except l stays the same, hence the constraint $x' = x \wedge s' = s \wedge n' = n$. The value of l, however, changes from 0 to 1, and the transition is only possible when $x < n$.

Program 3.1 defines the transition relation $\rho_{\mathit{Sum}}(l, x, s, n, l', x', s', n')$ defined as

$$\begin{aligned}
&(l = 0 \wedge x < n \wedge l' = 1 \wedge x' = x \wedge s' = s \wedge n' = n) \vee \\
&(l = 0 \wedge x \geq n \wedge l' = \Omega \wedge x' = x \wedge s' = s \wedge n' = n) \vee \\
&(l = 1 \wedge l' = 2 \wedge x' = x \wedge s' = s + x \wedge n' = n) \vee \\
&(l = 2 \wedge l' = 0 \wedge x' = x + 1 \wedge s' = s \wedge n' = n)
\end{aligned}$$

The set of reachable states of Program 3.1 is the condition

$$\begin{aligned}
&(l = 0 \wedge s = (x^2 - x)/2 \wedge 0 \leq x \leq n) \vee \\
&(l = 1 \wedge s = (x^2 - x)/2 \wedge 0 \leq x < n) \vee \\
&(l = 2 \wedge s = (x^2 + x)/2 \wedge 0 \leq x < n) \vee \\
&(l = \Omega \wedge s = (x^2 - x)/2 \wedge 0 \leq x = n),
\end{aligned}$$

The ancestor states of the exit condition $l = \Omega$ of Program 3.1 is characterized by the formula $l = \Omega \vee l = 0 \vee l = 1 \vee l = 2$. This means that the final program point is always reachable from any program point.

3.1.2 CLP Semantics

We start by defining a translation of a program in the syntax of Figure 3.1 into a CLP program. We first define the notion of enclosing statement of a program point l, denoted $\mathit{enclosing}(l)$ as follows:

- When l labels a statement inside a **then** or **else** block of an innermost if conditional labeled with l_{if}, then $\mathit{enclosing}(l) = l_{if}$. What we mean by "innermost" here is that there is no other if conditional or while loop between l from l_{if}. Hence, $\mathit{enclosing}(l)$ is uniquely determined.
- When l labels a statement inside a the body of an innermost while loop labeled with l_{while}, then $\mathit{enclosing}(l) = l_{while}$.
- Other than the two cases above, $\mathit{enclosing}(l) = \Omega$.

We define the notion of *next label* of a statement labeled with l, denoted $\mathit{next\_label}(l)$ as follows:

- $\mathit{next\_label}(\Omega) = \Omega$.
- When l labels a sequence of assignments or **skip** followed by another statement in a sequence, then $\mathit{next\_label}(l)$ is the program point of the next statement in the sequence. When there is no such statement, it is $\mathit{next\_label}(l) = \mathit{next\_label}(\mathit{enclosing}(l))$. If the next statement is **goto** $\langle m \rangle$ (note that according to our syntax in Figure 3.1, **goto** statements are not labeled), then $\mathit{next\_label}(l) = m$.

- When l labels an if conditional, it has three next labels:
  - $\mathit{next\_label}(l)$ is the program point of the first statement after the **end if** . If there is no such statement, it is $\mathit{next\_label}(\mathit{enclosing}(l))$.
  - $\mathit{next\_label\_then}(l)$ is the program point of the first statement in the **then** block. If the first statement in the **then** block is **goto** $\langle m \rangle$ then $\mathit{next\_label\_then}(l) = m$.
  - $\mathit{next\_label\_else}(l)$ is the program point of the first statement in the **else** block, if there is an **else** block. If the first statement is the **else** block is **goto** $\langle m \rangle$, then $\mathit{next\_label\_else}(l) = m$.

  We assume that there is no empty **then** or **else** block.

- When l labels a while loop, it also has three next labels:
  - $\mathit{next\_label}(l) = l$.
  - $\mathit{next\_label\_then}(l)$ is the program point of the first statement of the loop body. We assume that a loop body is never empty. Similar to the if conditional, if the first statement is **goto** $\langle m \rangle$, then $\mathit{next\_label\_then}(l) = m$.
  - $\mathit{next\_label\_else}(l)$ is the program point of the statement immediately following **end do**. If there is no such statement, it is $\mathit{next\_label}(\mathit{enclosing}(l))$. If the statement is **goto** $\langle m \rangle$, then $\mathit{next\_label\_else}(l) = m$.

Complementing the sequence $\tilde{x} = x_1, \ldots, x_n$ of n distinct program variables of a sequential program, we have a sequence $\tilde{X} = X_1, \ldots, X_n$ of distinct *CLP variables*, and a we draw a correspondence between program variable x_i and CLP variable X_i. We denote by $\tilde{X}'$ the sequence of the primed versions of the variables in $\tilde{X}$. We always use lower case letters to represent program variables, and sequence of characters with capital first letter for CLP variables.

We define a function trans_b which maps a sequential program written in our language into CLP program clauses as follows:

- $\mathit{trans}_b(\mathit{stmt}_1 \ldots \mathit{stmt}_k) = \left[\begin{array}{c} \mathit{trans}_b(\mathit{stmt}_1) \\ \vdots \\ \mathit{trans}_b(\mathit{stmt}_k). \end{array} \right.$

  Here, stmt_i is a labeled statement, possibly a **goto** statement.

- $\mathit{trans}_b($**goto** $\langle l \rangle)$ returns nothing.

- $trans_b(\langle l\rangle\ x_{i_1}\ :=\ expr_1 \ldots x_{i_q}\ :=\ expr_q) =$

$$\left[\begin{array}{l} p(next\_label(l), \tilde{X}^q)\ \texttt{:-}\ p(l,\tilde{X}), \\ \quad X_1^1 = X_1, \ldots, X_{i_1}^1 = expr_1\theta, \ldots, X_n^1 = X_n, \\ \qquad\qquad \ldots \\ \quad X_1^q = X_1^{q-1}, \ldots, X_{i_q}^q = expr_q\theta^{q-1}, \ldots, X_n^q = X_n^{q-1}. \end{array}\right.$$

  We denote by θ the renaming of program variables with the corresponding CLP variables. Also, we denote by θ^i the renaming of program variables with version i of the corresponding CLP variables. For example, when θ renames program variable x to CLP variable X, θ^i would renames program variable x to CLP variable X^i.

- $trans_b(\langle l\rangle$ **skip**$) =\ p(next\_label(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}).$

- $trans_b(\langle l\rangle$ **if** $(boolexpr)$ **then** $stmt_1 \ldots stmt_k$ **end if** $) =$

$$\left[\begin{array}{l} p(next\_label\_then(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), boolexpr\theta. \\ p(next\_label(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_k) \end{array}\right.$$

- $trans_b(\langle l\rangle$ **if** $(boolexpr)$ **then** $stmt_1 \ldots stmt_j$ **else** $stmt_{j+1} \ldots stmt_k$ **end if** $) =$

$$\left[\begin{array}{l} p(next\_label\_then(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), boolexpr\theta. \\ p(next\_label\_else(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_j)\ trans_b(stmt_{j+1} \ldots stmt_k) \end{array}\right.$$

- $trans_b(\langle l\rangle$ **while** $(boolexpr)$ **do** $stmt_1 \ldots stmt_k$ **end do** $) =$

$$\left[\begin{array}{l} p(next\_label\_then(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), boolexpr\theta. \\ p(next\_label\_else(l),\tilde{X})\ \texttt{:-}\ p(l,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_k) \end{array}\right.$$

As in Program 3.1, we often describe the initial state of the program using the clause "Initially." We translate "Initially *boolexpr*" as the CLP fact

$$p(l,\tilde{X})\ \ \texttt{:-}\ \ boolexpr\theta.$$

$$\begin{array}{ll}
p(0,0,0,N). & \kappa_1 \\
p(1,X,S,N) \;\texttt{:-}\; p(0,X,S,N), X < N. & \kappa_2 \\
p(\Omega,X,S,N) \;\texttt{:-}\; p(0,X,S,N), X \geq N. & \kappa_3 \\
p(2,X,S',N) \;\texttt{:-}\; p(1,X,S,N), S' = S+X. & \kappa_4 \\
p(0,X',S,N) \;\texttt{:-}\; p(2,X,S,N), X' = X+1. & \kappa_5
\end{array}$$

Program 3.2: Sum Backward CLP Model

Here, l is the initial program point.

We define the semantics of the original program to be its CLP model. In this definition, the interpretation of the predicate p in the least model of the CLP program is the set of reachable states of the program. This corresponds to the characterization of the set of descendant states of the set of entry states of a discrete dynamic system as a least fixpoint in [33].

Example 3.2. Sum The CLP model of the program Sum (Program 3.1) is Program 3.2. Note that the CLP variable X and its primed version corresponds to the program variable x, the CLP variable S corresponds to the program variable s, and the CLP variable N corresponds to the program variable n. The model is qualified as "backward" for a reason to be explained later.

The least model of the CLP Program 3.2 is

$$\begin{array}{lll}
\{p(\alpha,\beta,\gamma,\delta) & | & (\alpha = 0 \wedge \gamma = (\beta^2 - \beta)/2 \wedge 0 \leq \beta \leq \delta) \vee \\
& & (\alpha = 1 \wedge \gamma = (\beta^2 - \beta)/2 \wedge 0 \leq \beta < \delta) \vee \\
& & (\alpha = 2 \wedge \gamma = (\beta^2 + \beta)/2 \wedge 0 \leq \beta < \delta) \vee \\
& & (\alpha = \Omega \wedge \gamma = (\beta^2 - \beta)/2 \wedge 0 \leq \beta = \delta)\},
\end{array}$$

in which the interpretation of p exactly models the reachable states of the original Program 3.1 according to the usual semantics given in Example 3.1.

3.1.3 Forward CLP Model

So far our CLP model of sequential programs seems to be the "reverse" of the resolution step, because a CLP clause represents a transition from the body of a clause to the head of a clause. A resolution step, on the other hand, tries to unify with a head to obtain its body. We call this kind of representation as *backward* CLP model.

Of course, this suggests a *forward* CLP model, which can be obtained by translating differently from the original program. So instead of using the translation function $trans_b$ as before, we use a new translation function $trans_f$, defined as follows:

- $$trans_f(stmt_1 \ldots stmt_n) = \left[\begin{array}{c} trans_f(stmt_1) \\ \vdots \\ trans_f(stmt_n). \end{array} \right.$$
- $trans_f(\textbf{goto}\ \langle l \rangle)$ returns nothing.
- $trans_f(\langle l \rangle x_{i_1} \ := \ expr_1 \ldots x_{i_q} \ := \ expr_q) =$

$$\left[\begin{array}{l} p(next\_label(l), \tilde{X}) \ \texttt{:-}\ p(l, \tilde{X}^q), \\ \quad X_1^1 = X_1, \ldots, X_{i_1}^1 = expr_1\theta, \ldots, X_n^1 = X_n, \\ \qquad \ldots \\ \quad X_1^q = X_1^{q-1}, \ldots, X_{i_q}^q = expr_q\theta^{q-1}, \ldots, X_n^q = X_n^{q-1}. \end{array} \right.$$

- $trans_f(\langle l \rangle\ \textbf{skip}) = p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label(l), \tilde{X})$.
- $trans_f(\langle l \rangle\ \textbf{if}\ (boolexpr)\ \textbf{then}\ stmt_1 \ldots stmt_k\ \textbf{end if}\ ) =$

$$\left[\begin{array}{l} p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label\_then(l), \tilde{X}), boolexpr\theta. \\ p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label(l), \tilde{X}), \neg boolexpr\theta. \\ trans_f(stmt_1 \ldots stmt_k) \end{array} \right.$$

- $trans_f(\langle l \rangle\ \textbf{if}\ (boolexpr)\ \textbf{then}\ stmt_1 \ldots stmt_j\ \textbf{else}\ stmt_{j+1} \ldots stmt_k\ \textbf{end if}\ ) =$

$$\left[\begin{array}{l} p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label\_then(l), \tilde{X}), boolexpr\theta. \\ p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label\_else(l), \tilde{X}), \neg boolexpr\theta. \\ trans_f(stmt_1 \ldots stmt_j)\ trans_f(stmt_{j+1} \ldots stmt_k) \end{array} \right.$$

- $trans_f(\langle l \rangle\ \textbf{while}\ (boolexpr)\ \textbf{do}\ stmt_1 \ldots stmt_k\ \textbf{end do}\ ) =$

$$\left[\begin{array}{l} p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label\_then(l), \tilde{X}), boolexpr\theta. \\ p(l, \tilde{X}) \ \texttt{:-}\ p(next\_label\_else(l), \tilde{X}), \neg boolexpr\theta. \\ trans_f(stmt_1 \ldots stmt_k) \end{array} \right.$$

Compared to the definition of $trans_b$, it is easy to see in $trans_f$ we simply exchange the predicate p in the head of the clause with the one in the body of the clause.

$$\begin{array}{lll} p(\Omega,X,S,N). & & \kappa_6 \\ p(0,X,S,N) & \texttt{:-}\ p(1,X,S,N), X < N. & \kappa_7 \\ p(0,X,S,N) & \texttt{:-}\ p(\Omega,X,S,N), X \geq N. & \kappa_8 \\ p(1,X,S,N) & \texttt{:-}\ p(2,X,S',N), S' = S+X. & \kappa_9 \\ p(2,X,S,N) & \texttt{:-}\ p(2,X,S,N), X' = X+1. & \kappa_{10} \end{array}$$

Program 3.3: Sum Forward CLP Model (Repeat of Figure 1.2)

In the forward translation, we do not translate the initial state a clause. Instead, we translate the condition of interest. Suppose that our condition of interest is the constraint c on the program variables. Then we include the clause

$$p(l,\tilde{X})\ \texttt{:-}\ c\theta.$$

Here, l is the program point of interest.

The least model of the forward CLP model of a program corresponds to the set of ancestors of the condition of interest. When the condition of interest is the final program label, this is equivalent to the set of ancestors of the exit states in [33].

Example 3.3. Program 3.3 is the forward CLP model of Program 3.1. Here the condition of interest is $l = \Omega$.

The least model of Program 3.3 is

$$\{p(\alpha,\beta,\gamma,\delta) \quad | \quad \alpha = \Omega \vee \alpha = 0 \vee \alpha = 1 \vee \alpha = 2\}.$$

This corresponds to the set of ascendant states of the condition of interest in Example 3.1.

3.1.4 Final Variables

Often our objective in analyzing a program is to reason about the values of variables whenever a program point of interest is reached from the initial state of the program. This is easily done with backward CLP model of a program, since the least model of the CLP program represents all the reachable states of the program.

The situation is not as simple with the forward model, since the least model of the CLP program corresponds to states that can possibly reach the condition of interest. We can, however reason that the condition of interest is reachable from the initial state by showing that a represen-

$$\begin{array}{lll l}
p(\Omega,X,S,N,S). & & & \kappa_1 \\
p(0,X,S,N,S_f) & \text{:-} & p(1,X,S,N,S_f),X<N. & \kappa_2 \\
p(0,X,S,N,S_f) & \text{:-} & p(\Omega,X,S,N,S_f),X\geq N. & \kappa_3 \\
p(1,X,S,N,S_f) & \text{:-} & p(2,X,S',N,S_f),S'=S+X. & \kappa_4 \\
p(2,X,S,N,S_f) & \text{:-} & p(0,X,S,N,S_f),X'=X+1. & \kappa_5
\end{array}$$

Program 3.4: Sum Forward CLP Model with Final Variables

tation of the initial state of the program is included in the least model of the CLP program. To check that the program point is always reached with the correct variable values, we need to make explicit in the least model the variable values whenever the condition of interest is reached. For this purpose, we add a sequence of variables called *final variables*, which are copies of variables of interest at the point of interest. Whenever Y denotes a representation of program variable y, Y_f denotes the final variable version of Y. Similarly, whenever $\tilde{X}$ denotes the sequence of the representation of program variables, $\tilde{X}_f$ denotes the final version of the sequence.

Recall that in a forward model, a clause represents an execution of a labeled program statement, and the constraint fact represents the condition of interest. The final variables are not touched in the clauses, that is they are copied as is, from the arguments of the p predicate of the head to the same predicate in the body, without being referred to in other parts of the clause. We only require that at the constraint fact representing the condition of interest, they are unified with program variable representations. In this way, the least model of the CLP program interprets the predicate $p(L,\tilde{X},\tilde{X}_f)$, which is true under ground substitution σ if $\tilde{X}_f\sigma$ is the value of the program variables at the condition of interest, when the condition of interest is reached from program point $L\sigma$ with program variable values $\tilde{X}\sigma$.

Program 3.4 is the forward CLP model of Program 3.1 with a final variable S_f, which is the final version of S. In the example we do not provide the final versions of other variables. We usually only provide the final versions of a subset of program variable representations that are essential for the reasoning.

3.1.5 Programs with Array

In place of variables, we may have array references in the program. In our CLP model we use a variable to denote any array that is used in the program. Array reference $a[i]$ is straightforwardly the expression renamed to $A[I]$ by the renaming θ in the CLP model, where I is the renaming of i, and A is the renaming of a in the CLP model. A is the CLP variable which denotes the array variable *a* in the program. A special note is with assignments of the form

$$a[i] \ := \ expr$$

which may appear in the middle of a sequence of assignments. This we translate using array update expression into $A' = \langle A, I, expr\rangle$ in the CLP program. More precisely,
$trans_b(\langle l\rangle x_{i_1} \ := \ expr_1 \ldots x_{i_r}[x_t] \ := \ expr_r \ldots x_{i_q} \ := \ expr_q) =$

$$\left[\begin{array}{l} p(next\_label(l), \tilde{X}^q) \ \texttt{:-} \ p(l, \tilde{X}), \\ \quad X_1^1 = X_1, \ldots, X_{i_1}^1 = expr_1\theta, \ldots, X_n^1 = X_n, \\ \qquad \ldots \\ \quad X_1^r = X_1^{r-1}, \ldots, X_{i_r}^r = \langle X_{i_r}^{r-1}, X_t^{r-1}, expr_r\theta^{r-1}\rangle, \ldots, X_n^r = X_n^{r-1}, \\ \qquad \ldots \\ \quad X_1^q = X_1^{q-1}, \ldots, X_{i_q}^q = expr_q\theta^{q-1}, \ldots, X_n^q = X_n^{q-1}. \end{array}\right.$$

Similarly, $trans_f(\langle l\rangle x_{i_1} \ := \ expr_1 \ldots x_{i_r}[x_t] \ := \ expr_r \ldots x_{i_q} \ := \ expr_q) =$

$$\left[\begin{array}{l} p(l, \tilde{X}) \ \texttt{:-} \ p(next\_label(l), \tilde{X}^q), \\ \quad X_1^1 = X_1, \ldots, X_{i_1}^1 = expr_1\theta, \ldots, X_n^1 = X_n, \\ \qquad \ldots \\ \quad X_1^r = X_1^{r-1}, \ldots, X_{i_r}^r = \langle X_{i_r}^{r-1}, X_t^{r-1}, expr_r\theta^{r-1}\rangle, \ldots, X_n^r = X_n^{r-1}, \\ \qquad \ldots \\ \quad X_1^q = X_1^{q-1}, \ldots, X_{i_q}^q = expr_q\theta^{q-1}, \ldots, X_n^q = X_n^{q-1}. \end{array}\right.$$

As an example, consider Program 3.5 and its forward CLP model Program 3.6. In our CLP model, we eliminate the representation of the program variable t, since it is only used internally in the assignment sequence starting at $\langle 5\rangle$.

3.1.6 Programs with Heap and Recursive Pointer Data Structures

We may also model in CLP a program which manipulates pointer-based data structures, such as a linked list and a binary tree. We allow structure member references *x*→*val*, *x*→*next*, *x*→*left*, *x*→*right* in place of normal variables in our simple programming language. The informal semantics is that *x* is a pointer variable pointing to a structure with members *val*, *next*, *left* or *right*. Now, *x*→*member* is the value of the element *member* of the structure (which is any of *val*, *next*,

```
⟨0⟩  $i := 0$
⟨1⟩  **while** $(i < n-1)$ **do**
⟨2⟩    $j := 0$
⟨3⟩    **while** $(j < n-1-i)$ **do**
⟨4⟩      **if** $(a[j+1] \leq a[j])$ **then**
⟨5⟩        $t := a[j+1]$
           $a[j+1] := a[j]$
           $a[j] := t$
         **end if**
⟨6⟩      $j := j+1$
       **end do**
⟨7⟩    $i := i+1$
     **end do**
```

Program 3.5: Bubble Sort

$$
\begin{array}{llll}
p(\Omega,A,I,J,N,A,N). & & & \kappa_1 \\
p(0,A,I,J,N,A_f,N_f) & \text{:-} & p(1,A,0,J,N,A_f,N_f). & \kappa_2 \\
p(1,A,I,J,N,A_f,N_f) & \text{:-} & I \geq N-1, p(\Omega,A,I,J,N,A_f,N_f). & \kappa_3 \\
p(1,A,I,J,N,A_f,N_f) & \text{:-} & I < N-1, p(2,A,I,J,N,A_f,N_f). & \kappa_4 \\
p(2,A,I,J,N,A_f,N_f) & \text{:-} & p(3,A,I,0,N,A_f,N_f). & \kappa_5 \\
p(3,A,I,J,N,A_f,N_f) & \text{:-} & J \geq N-1-I, p(7,A,I,J,N,A_f,N_f). & \kappa_6 \\
p(3,A,I,J,N,A_f,N_f) & \text{:-} & J < N-1-I, p(4,A,I,J,N,A_f,N_f). & \kappa_7 \\
p(4,A,I,J,N,A_f,N_f) & \text{:-} & A[J+1] > A[J], p(6,A,I,J,N,A_f,N_f). & \kappa_8 \\
p(4,A,I,J,N,A_f,N_f) & \text{:-} & A[J+1] \leq A[J], p(5,A,I,J,N,A_f,N_f). & \kappa_9 \\
p(5,A,I,J,N,A_f,N_f) & \text{:-} & A' = \langle\langle A,J+1,A[J]\rangle,J,A[J+1]\rangle, & \\
\quad p(6,A',I,J,N,A_f,N_f). & & & \kappa_{10}
\end{array}
$$

Program 3.6: Bubble Sort Forward CLP Model

left or *right*).

In order to translate member references, we need to note that they actually refer to the program heap, albeit implicitly. The program heap itself can be modeled as an array, which we name using an auxiliary variable *h*. This array is indexed by pointer variables. When *x* is the address of the structure, we assume that the member *val* is always stored at address *x*, *next* or *left* at the address $x+1$, and *right* at address $x+2$. We then provide the following alternatives to member references[1]:

[1]The use of array to denote structure member reference here is following Reynolds [166].

```
Initially $p \neq 0$.
⟨0⟩   while  $(p \neq 0)$ do
⟨1⟩       $p \rightarrow val$ := 0
⟨2⟩       $p$ := $p \rightarrow next$
      end do
```

Program 3.7: List Elements Reset

$$
\begin{array}{lll}
p(0,H,P,H_f,P_f) & :\text{-} & P = 0, p(\Omega,H,P,H_f,P_f). \\
p(0,H,P,H_f,P_f) & :\text{-} & P \neq 0, p(1,H,P,H_f,P_f). \\
p(1,H,P,H_f,P_f) & :\text{-} & p(2,\langle H,P,0\rangle,P,H_f,P_f). \\
p(2,H,P,H_f,P_f) & :\text{-} & p(0,H,H[P+1],H_f,P_f). \\
p(\Omega,H,P,H,P). & &
\end{array}
$$

Program 3.8: List Elements Reset CLP Model

Member Reference	**Alternative Expression**
$x \rightarrow val$	$h[x]$
$x \rightarrow next$	$h[x+1]$
$x \rightarrow left$	$h[x+1]$
$x \rightarrow right$	$h[x+2]$

Notice that both $x \rightarrow next$ and $x \rightarrow left$ have the same alternative expression. This is never ambiguous, since they have different use. We use the member *next* to denote the address of the next structure in a linked list, while *left* is used to denote the address of the left child structure of a binary tree.

An assignment to structure member such as $x \rightarrow val$:= $expr$, therefore has an alternative $h[x]$:= $expr$. Expressions containing member reference as well as assignments to structure member are therefore modeled in CLP as we would model array references and assignments to array elements explained in the previous section. As an example, Program 3.7 which resets all the values of a linked list into 0, is modeled in CLP, using forward modeling, as Program 3.8.

Known programming languages have ways to allocate some area of the heap. For this purpose, we add the structure allocation expression

$$\mathbf{new}(expr_1, expr_2, expr_3)$$

into our simple programming language. It simply denotes an unspecified address, say *new*, of the heap, where $h[new] = expr_1$, $h[new+1] = expr_2$, and $h[new+2] = expr_3$. Or we may also use the simpler

$$\mathbf{new}(expr_1, expr_2)$$

which similarly denotes an unspecified address, where $h[new] = expr_1$ and $h[new+1] = expr_2$.

Without being too formal, here we simply say that to translate allocation expression into CLP, we extend θ such that $(\mathbf{new}(expr_1, expr_2, expr_3))\theta = New_i$, when $\mathbf{new}(expr_1, expr_2, expr_3)$ is the *i*-th allocation expression in a labeled statement. Then in the body of the clause modeling the statement, we add the constraints $H[New_i] = expr_1\theta$, $H[New_i + 1] = expr_2\theta$, and $H[New_i + 2] = expr_3\theta$. Of course, θ is the renaming to the appropriate variable version, when the labeled statement is a sequence of assignments (see Section 3.1.2). We also add the constraint $New_i \neq New_j$ for all $1 \leq j < i$ to declare separation between allocations. In addition, we also state that the new variable New_i is not shared with existing data structures by adding the atom $no\_reach(H, New_i, X)$, whenever X represents a pointer variable x in the program. The definition of *no_reach* depends on the data structure rooted at x.

The binary search tree value insertion program shown as Program 3.9 is an example of a program which uses heap allocation expression. The CLP model is shown as the CLP Program 3.10. The definition of the *no_reach* predicate that we use here is given as Program 3.11. The definition ensures that *New* is not shared with any cell of the tree rooted at X.

3.2 Multiprocedure Programs

In this section we discuss how a multiprocedure and multifragment programs can be translated into forward CLP models (translation into backward CLP models can be defined similarly).

In order to write multiprocedure programs, we need to define a few language constructs. They are:

1. *Procedure definitions* of the syntax **proc** *ProcName* (*VarSeq*) *Prg* **end proc**. Here, the tuple of formal arguments *VarSeq* is optional, and *Prg* is a program as defined earlier. *Procname* is the name of the procedure.

2. *Procedure call* of the syntax *ProcName* (*ExprSeq*) or x := *ProcName* (*ExprSeq*). The tuple (*ExprSeq*) is optional, depending on whether the procedure *ProcName* has formal arguments or not.

3. *Return statement* of the syntax **return** *Expr*, where *Expr* is optional.

```
Initially $x \neq 0$.
$\langle 0 \rangle$  **if** $(a < x{\rightarrow}val)$ **then**
$\langle 1 \rangle$      **if** $(x{\rightarrow}left = 0)$ **then**
$\langle 2 \rangle$          $x{\rightarrow}left$ := **new**$(a,0,0)$
                             $x$ := $x{\rightarrow}left$
                         **else**
$\langle 3 \rangle$          $x$ := $x{\rightarrow}left$
                             **goto** $\langle 0 \rangle$
                         **end if**
                     **else**
$\langle 5 \rangle$      **if** $(a > x{\rightarrow}val)$ **then**
$\langle 6 \rangle$          **if** $(x{\rightarrow}right = 0)$ **then**
$\langle 7 \rangle$              $x{\rightarrow}right$ := **new**$(a,0,0)$
                                 $x$ := $x{\rightarrow}right$
                             **else**
$\langle 8 \rangle$              $x$ := $x{\rightarrow}right$
                                 **goto** $\langle 0 \rangle$
                             **end if**
                         **end if**
                     **end if**
```

Program 3.9: Binary Search Tree Insertion

Defining a program as a procedure with name $\mathit{ProcName} = \mathit{procname}$ simply restricts the CLP model of its translation to use *procname* as predicate name, instead of the generic "*p*" we have been using earlier. The tuple of formal arguments (VarSeq) is important in a procedure call.

In procedural programming languages, some variables can be *local* to a procedure, some can be *global*. Some of the local variables can be considered as *argument* variables, including the variables that represent its formal arguments, and a *return value variable*. A return value variable of a procedure $\mathit{procname}_i$ is named r_i. We assume that all global variables $\tilde{g}$ are known, similarly all local variables $\tilde{x}_i$ of each procedure $\mathit{procname}_i$ that are not its formal arguments nor its return value variables (in real compilers, this information is obtained in a separate compilation phase).

We now describe the CLP semantics of a multiprocedure program starting with procedure definitions. A program now consists of a sequence of procedure definitions. In our framework, procedure definitions have no representations as CLP clauses: They simply define the clauses' predicate names and a set of local variables which are formal arguments of the procedures. A

$$
\begin{array}{lll}
p(0,H,X,A,H_f,X_f) & \text{:-} & p(1,H,X,A,H_f,X_f), A < H[X]. \\
p(0,H,X,A,H_f,X_f) & \text{:-} & p(5,H,X,A,H_f,X_f), A \geq H[X]. \\
p(1,H,X,A,H_f,X_f) & \text{:-} & p(2,H,X,A,H_f,X_f), H[X+1] = 0. \\
p(1,H,X,A,H_f,X_f) & \text{:-} & p(3,H,X,A,H_f,X_f), H[X+1] \neq 0. \\
p(2,H,X,A,H_f,X_f) & \text{:-} & p(\Omega,\langle H,X+1,New\rangle,New,A,H_f,X_f), \\
\quad H[New] = A, H[New+1] = 0, H[New+2] = 0, no\_reach(H,New,X). && \\
p(3,H,X,A,H_f,X_f) & \text{:-} & p(0,H,H[X+1],A,H_f,X_f). \\
p(5,H,X,A,H_f,X_f) & \text{:-} & p(6,H,X,A,H_f,X_f), A > H[X]. \\
p(5,H,X,A,H_f,X_f) & \text{:-} & p(\Omega,H,X,A,H_f,X_f), A \leq H[X]. \\
p(6,H,X,A,H_f,X_f) & \text{:-} & p(7,H,X,A,H_f,X_f), H[X+2] = 0. \\
p(6,H,X,A,H_f,X_f) & \text{:-} & p(8,H,X,A,H_f,X_f), H[X+2] \neq 0. \\
p(7,H,X,A,H_f,X_f) & \text{:-} & p(\Omega,\langle H,X+2,New\rangle,New,A,H_f,X_f), \\
\quad H[New] = A, H[New+1] = 0, H[New+2] = 0, no\_reach(H,New,X). && \\
p(8,H,X,A,H_f,X_f) & \text{:-} & p(0,H,H[X+2],A,H_f,X_f). \\
p(\Omega,H,X,A,H,X). &&
\end{array}
$$

Program 3.10: Binary Search Tree Insertion CLP Model

$$
\begin{array}{lll}
no\_reach(H,I,L) & \text{:-} & L = 0. \\
no\_reach(H,I,L) & \text{:-} & L \neq 0, I \neq L, \\
\quad no\_reach(H,I,H[L+1]), && \\
\quad no\_reach(H,I,H[L+2]). &&
\end{array}
$$

Program 3.11: *No_reach* for Binary Tree

multiprocedure program is translated into CLP model using $mptrans_f$ function below:

$$
\begin{array}{l}
mptrans_f(\textbf{proc}\ procname_1\ (\tilde{v}_1)\ body_1\ \textbf{end proc} \\
\qquad\vdots \\
\qquad \textbf{proc}\ procname_n\ (\tilde{v}_n)\ body_n\ \textbf{end proc}) = \\
bodytrans_f(procname_1(\tilde{v}_1), body_1) \\
\qquad\vdots \\
bodytrans_f(procname_n(\tilde{v}_n), body_n)
\end{array}
$$

The body of each procedure is translated into CLP using $bodytrans_f$ function. This function is essentially the $trans_f$ function we discussed in Section 3.1.3 but with specified predicate name and unique set of variables including global, formal arguments, local, return values, and final variables (Section 3.1.4) to represent updates to global variables. Given a procedure $procname_i$, we devise a mapping $\theta = \{\tilde{g} \mapsto \tilde{G}, \tilde{v}_i \mapsto \tilde{V}_i, \tilde{x}_i \mapsto \tilde{X}_i\}$ which maps global variables $\tilde{g}$, formal arguments $\tilde{v}_i$, and local variables $\tilde{x}_i$ into distinct CLP variables.

Now, the procedure calls are translated into CLP as follows:

```
          proc main
⟨0⟩          $t := a \times b$
⟨1⟩          $p$
⟨2⟩          $t := a \times b$
          end proc

          proc $p$
⟨0⟩          if $(a = 0)$ then
⟨1⟩              return
             else
⟨2⟩              $a := a - 1$
⟨3⟩              $p$
⟨4⟩              $t = a \times b$
             end if
          end proc
```

Program 3.12: Multiprocedure Program

$$\begin{array}{lll}
main(0,T,A,B,T_f,A_f,B_f) & \text{:-} & main(1,A\times B,A,B,T_f,A_f,B_f).\\
main(1,T,A,B,T_f,A_f,B_f) & \text{:-} & p(0,T,A,B,T',A',B'),main(2,T',A',B',T_f,A_f,B_f).\\
main(2,T,A,B,T_f,A_f,B_f) & \text{:-} & main(\Omega,A\times B,A,B,T_f,A_f,B_f).\\
main(2,T,A,B,T,A,B). & & \\
 & & \\
p(0,T,A,B,T_f,A_f,B_f) & \text{:-} & p(1,T,A,B,T_f,A_f,B_f),A=0.\\
p(0,T,A,B,T_f,A_f,B_f) & \text{:-} & p(2,T,A,B,T_f,A_f,B_f),A\neq 0.\\
p(1,T,A,B,T_f,A_f,B_f) & \text{:-} & p(\Omega,T,A,B,T_f,A_f,B_f).\\
p(2,T,A,B,T_f,A_f,B_f) & \text{:-} & p(3,T,A-1,B,T_f,A_f,B_f).\\
p(3,T,A,B,T_f,A_f,B_f) & \text{:-} & p(0,T,A,B,T',A',B'),p(4,T',A',B',T_f,A_f,B_f).\\
p(4,T,A,B,T_f,A_f,B_f) & \text{:-} & p(\Omega,T',A,B,T_f,A_f,B_f),T'=A\times B.\\
p(\Omega,T,A,B,T_f,A_f,B_f). & &
\end{array}$$

Program 3.13: Multiprocedure Program Forward CLP Model

- $bodytrans_f(procname_i(\tilde{v}_i), \langle l\rangle\ procname_j\ (\tilde{expr})) =$

$$\begin{array}{l}
procname_i(l,\tilde{G},\tilde{V}_i,\tilde{X}_i,R_i,\tilde{G}_f)\ \text{:-}\\
\quad procname_j(\tilde{G},\tilde{expr}\theta,\tilde{X}_j,R_j,\tilde{G}'),procname_i(next\_label(l),\tilde{G}',\tilde{V}_i,\tilde{X}_i,R_i,\tilde{G}_f).
\end{array}$$

- $bodytrans_f(procname_i(\tilde{v}_i), \langle l\rangle\ x\ :=\ procname_j\ (\tilde{expr})) =$

$$\begin{array}{l}
procname_i(l,\tilde{G},\tilde{V}_i,\tilde{X}_i,R_i,\tilde{G}_f)\ \text{:-}\\
\quad procname_j(0,\tilde{G},\tilde{expr}\theta,\tilde{X}_j,X',\tilde{G}'),procname_i(next\_label(l),\tilde{G}'',\tilde{V}_i',\tilde{X}_i',R_i,\tilde{G}_f).
\end{array}$$

Note that here either $X' \in \tilde{G}''$, $x' \in \tilde{V}_i'$, or $X' \in \tilde{X}_i'$. Also, $\tilde{G}'' = \tilde{G}'$ when $X' \notin \tilde{G}'$, $\tilde{V}_i' = \tilde{V}_i$

when $X' \notin \tilde{V}_i'$, and $\tilde{X}_i' = \tilde{X}_i$ when $X' \notin \tilde{X}_i'$.

Return statements transfer control to program point Ω. We require that all procedures that are called using the assignment form of the procedure call cannot reach program point Ω without having **return** *expr* as the last statement executed. **return** statements are translated into CLP as follows:

- $bodytrans_f(procname_i(\tilde{v}_i), \langle l \rangle$ **return** $) =$

$$\begin{aligned} & procname_i(l, \tilde{G}, \tilde{V}_i, \tilde{X}_i, R_i, \tilde{G}_f) \ \text{:-} \\ & \quad procname_i(\Omega, \tilde{G}, \tilde{V}_i, \tilde{X}_i, R_i, \tilde{G}_f). \end{aligned}$$

- $bodytrans_f(procname_i(\tilde{v}_i), \langle l \rangle$ **return** $expr) =$

$$\begin{aligned} & procname_i(l, \tilde{G}, \tilde{V}_i, \tilde{X}_i, R_i, \tilde{G}_f) \ \text{:-} \\ & \quad procname_i(\Omega, \tilde{G}, \tilde{V}_i, \tilde{X}_i, R_i, \tilde{G}_f), R_i = expr\theta. \end{aligned}$$

The discussion in this section together with the statechart example later in Section 3.7 demonstrate that CLP models are easily tailored to express compositional structure of the program to be modeled.

Example 3.4. We take the multiprocedure program of Sharir and Pnueli [179] as an example. The program is shown as Program 3.12, with its CLP model Program 3.13. The program only has global variables t, a, and b, and both procedures *main* and p have no formal arguments nor return values.

In this thesis we are somewhat liberal in our translation into CLP models. In Program 3.13 we simplify our model not to include return value variables. Also, we have $\langle 2 \rangle$ as the program point of interest in the procedure *main* instead of Ω. This is for our verification purpose later in Chapter 5.

3.3 Concurrent Programs

Concurrency is in essence nondeterminism [5], and CLP clauses are suitable for representing nondeterministic transition systems. In this section we show how to model concurrent programs in CLP.

3.3.1 Syntax

Concurrency often coincides with programs that run forever. We enclose such program in a **loop forever** ... **end loop** construct. Instead of Ω, the next label of the last statement of the program is the program point of its first statement. For the more formal translation into CLP semantics, we can mostly still use the *enclosing* and *next_label* definitions as in the previous section. However, we redefine $\mathit{next\_label}(\Omega) = l$, where l is the program point of the first statement of the program.

Now, in a concurrent setting we call each program a *process*, and a *concurrent program* consists of more than one processes. Whereas in a sequential setting a program is executed without interruption until it is (hopefully) terminated, in concurrent setting, an operating system running in the background may stop a running process, and execute another process. We assume that a process may only be stopped after it has completely executed a statement, and hence for concurrent programs we adopt the so-called *asynchronous* or *interleaving* semantics. Later in Section 3.6 we will also demonstrate the modeling of the complementary *synchronous* concurrency where transitions of different processes (automata) are executed at the same time.

Often in a concurrent setting, processes cooperate with one another. For this purpose, they need to communicate, and communication is only possible if at least one party waits for information from others. We realize the waiting of a process by introducing an *await* statement of the syntax

$$\textbf{await}\ (\mathit{BoolExpr})$$

or

$$\textbf{await}\ (\mathit{BoolExpr})\ \mathit{Variable}\ := \ \mathit{Expr}$$

Upon reaching any of the above statement, a process may only progress to the next program point when the given boolean expression is true. With the latter syntax, when the boolean expression is true, the given assignment is first executed before control progresses to the next program point.

Example 3.5. (Bakery Algorithm) Now consider our specification of two-process Bakery mutual exclusion algorithm [125] as shown in Program 3.14. In the program, variables x and y denote the “ticket numbers” of each of Process 1 and 2, respectively. They are both 0 before the program runs. Whenever a process is interested to enter its critical section (program point 2), it sets its ticket number to one more than the other process’ ticket number.

The program variables of a concurrent program includes variables in the program text, and the variables $l_1,\ldots,l_n$, to store the program point of each process i, $1 \leq i \leq n$. The state of a concurrent program is thus a ground substitution of $l_1,\ldots,l_n$ and the variables of program text into constants in the corresponding domains.

3.3.2 CLP Semantics

Interleaving semantics means that at any one time, only a statement in any of the processes can be executing. Therefore, a state transition of a concurrent program represents only one state transition in any of its processes. We perform a translation of a process i, where $1 \leq i \leq m$, where m is the total number of processes, and n is the total number of variables, into backward CLP model as follows:

- $trans_b(stmt_1 \ldots stmt_k) = trans_b(stmt_1) \ldots trans_b(stmt_k)$. Here, $stmt_i$ is a labeled statement, possibly a **goto** statement.
- $trans_b($**goto** $\langle l_i \rangle)$ returns nothing.
- $trans_b(\langle l_i \rangle x_j \ := \ expr) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label(l_i),\ldots,l_m,\tilde{X}') \text{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), X'_1 = X_1,\ldots,X'_j = expr\theta,\ldots,X'_n = X_n. \end{array}\right.$$

- $trans_b(\langle l_i \rangle$ **await** $(boolexpr)) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label(l_i),\ldots,l_m,\tilde{X}') \text{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), boolexpr\theta. \end{array}\right.$$

- $trans_b(\langle l_i \rangle$ **await** $(boolexpr)\ x_j \ := \ expr) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label(l_i),\ldots,l_m,\tilde{X}') \text{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), \\ \quad boolexpr\theta, X'_1 = X_1,\ldots,X'_j = expr\theta,\ldots,X'_n = X_n. \end{array}\right.$$

- $trans_b(\langle l_i \rangle$ **if** $(boolexpr)$ **then** $stmt_1 \ldots stmt_k$ **end if** $) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label\_then(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), boolexpr\theta. \\ p(l_1,\ldots,next\_label(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_k) \end{array}\right.$$

- $trans_b(\langle l_i \rangle$ **if** $(boolexpr)$ **then** $stmt_1 \ldots stmt_j$ **else** $stmt_{j+1} \ldots stmt_k$ **end if** $) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label\_then(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), boolexpr\theta. \\ p(l_1,\ldots,next\_label\_else(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_j)\ trans_b(stmt_{j+1} \ldots stmt_k) \end{array}\right.$$

- $trans_b(\langle l_i \rangle$ **while** $(boolexpr)$ **do** $stmt_1 \ldots stmt_k$ **end do** $) =$

$$\left[\begin{array}{l} p(l_1,\ldots,next\_label\_then(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), boolexpr\theta. \\ p(l_1,\ldots,next\_label\_else(l_i),\ldots,l_m,\tilde{X}) \texttt{ :- } \\ \quad p(l_1,\ldots,l_i,\ldots,l_m,\tilde{X}), \neg boolexpr\theta. \\ trans_b(stmt_1 \ldots stmt_k) \end{array}\right.$$

As in Program 3.1, we often describe the initial state of the program using the clause "Initially." We translate "Initially *boolexpr*" as the CLP fact

$$p(l_1,\ldots,l_m,\tilde{X}) \texttt{ :- } boolexpr\theta.$$

Here, $l_1,\ldots,l_m$ are the initial program points of process 1 to m.

```
Initially $x = 0$ and $y = 0$.
Process 1                                  Process 2
      loop forever                               loop forever
$\langle 0 \rangle$    $x := y+1$                     $\langle 0 \rangle$    $y := x+1$
$\langle 1 \rangle$    await $(x < y \vee y = 0)$     $\langle 1 \rangle$    await $(y < x \vee x = 0)$
$\langle 2 \rangle$    $x := 0$                       $\langle 2 \rangle$    $y := 0$
      end loop                                   end loop
```

Program 3.14: Two-Process Bakery Algorithm

```
$p(0,0,0,0).$                                                $\kappa_1$
$p(1,L_2,Y+1,Y)$ :- $p(0,L_2,X,Y).$                          $\kappa_2$
$p(2,L_2,X,Y)$ :- $p(1,L_2,X,Y),(Y = 0 \vee X < Y).$         $\kappa_3$
$p(0,L_2,0,Y)$ :- $p(2,L_2,X,Y).$                            $\kappa_4$
$p(L_1,1,X,X+1)$ :- $p(L_1,0,X,Y).$                          $\kappa_5$
$p(L_1,2,X,Y)$ :- $p(L_1,1,X,Y),(X = 0 \vee Y < X).$         $\kappa_6$
$p(L_1,0,X,0)$ :- $p(L_1,2,X,Y).$                            $\kappa_7$
```

Program 3.15: Two-Process Bakery Algorithm CLP Model

The backward CLP model of Program 3.14 is Program 3.15. The state space of Program 3.14 is given by the least model of the CLP Program 3.15, which is as follows:

$$\begin{aligned}\{p(L_1,L_2,X,Y) \quad | \quad & (L_1 = 0 \wedge 0 \leq L_2 \leq 2 \wedge X = 0 \wedge Y \geq 0) \vee \\ & (0 \leq L_1 \leq 2 \wedge L_2 = 0 \wedge X \geq 0 \wedge Y = 0) \vee \\ & (L_1 = 1 \wedge (L_2 = 1 \vee L_2 = 2) \wedge X = Y+1) \vee \\ & ((L_1 = 1 \vee L_1 = 2) \wedge L_2 = 1 \wedge Y = X+1)\}.\end{aligned}$$

Clearly, the least model is infinite, that is, there are infinite ground substitution σ such that $p(L_1,L_2,X,Y)\sigma$ is in the above least model. This means that the Bakery Algorithm is an *infinite-state program*.

In the next sections we show how we can add more details into our modeling of concurrent programs in CLP.

3.3.3 Scheduling

Concurrent programs are often controlled by an operating system, which schedules the processes. Here we demonstrate via an example how we may embed the operating system's scheduling policy in the CLP model.

Consider a simple two-process concurrent program shown as Program 3.16. We wish to add

```
Initially $x = 0$ and $y = 0$.
Process 1                          Process 2
      loop forever                       loop forever
$\langle 0 \rangle$      $x := x+1$              $\langle 0 \rangle$      $y := y+1$
      end loop                           end loop
```

Program 3.16: Scheduled Concurrent Program

```
$p(0,0,Q,X,Y)$ :- $Q = 0, X = 0, Y = 0$.
$p(0,L_2,Q+1,X+1,Y)$ :- $Q \leq 2, p(0,L_2,Q,X,Y)$.
$p(L_1,0,0,X,Y+1)$ :- $Q > 0, p(L_1,0,Q,X,Y)$.
```

Program 3.17: Scheduled Concurrent Program CLP Model

the scheduling policy where Process 1 executes at least one and at most three statements before control is passed to Process 2. Thus we implement a *k-fair scheduler* where $k = 3$ in this case.

We include this scheduling policy in our backward CLP model shown as Program 3.17. The program is translated as described in Section 3.3.2, but here we add a variable Q representing the state of the scheduler. Q is incremented whenever Process 1 executes, but Process 1 can only execute while $Q \leq 2$. On the other hand, execution of Process 2 is only possible when $Q > 0$, that is when Process 1 has been executed after the last execution of Process 2, and this resets Q to 0.

3.4 Timed Programs

Program 3.18 is a concurrent program that computes the Fibonacci numbers and assign them to the array a such that $a[x]$ contains the x-th Fibonacci number. Both processes are run on separate processors, but they access the shared variables x, y and the array a. We take the liberty of introducing a new syntax: **delay** (t), which informal semantics is to delay the program t time units. During the delay, the program do not access any of the program variables.

Now we provide informal explanation of the processes. Process 1 assigns on the array a's even indices x the x-th Fibonacci number, while Process 2 does the same with odd indices. There is a danger that either the assignment at program point $\langle 2 \rangle$ of Process 1 or the program point $\langle 3 \rangle$ of Process 2 may refer to an array element that has not been assigned a Fibonacci number. Here the system performs no scheduling, but with the right timings, the program remains correct to an extent. We assume that every program statement takes a fixed number ε of time units, where $95 \leq \varepsilon \leq 105$.

Initially $a[0] = 0$, $a[1] = 1$ and $a[i] = 0$ for all $i \geq 2$.

```
Process 1                              Process 2
⟨0⟩  $x := 2$                           ⟨0⟩  $y := 3$
⟨1⟩  while $(x \leq n)$ do              ⟨1⟩  delay(300)
⟨2⟩    $a[x] := a[x-1] + a[x-2]$        ⟨2⟩  while $(y \leq n)$ do
⟨3⟩    $x := x + 2$                     ⟨3⟩    $a[y] := a[y-1] + a[y-2]$
     end do                            ⟨4⟩    $y := y + 2$
                                            end do
```

Program 3.18: Dangerous Parrallel Fibonacci with Fixed Timing

The backward CLP model is the Program 3.19. In the CLP model we add the auxiliary variables T_1 and T_2, which we call *clock variables*. Without further ado, we assume that the clock variables have as their domain the set of real numbers. This domain can be easily included among the domains already introduced in Chapter 2[2]. T_1 denotes how much time the processor dedicated to Process 1 has spent in executing Process 1. Similarly, T_2 denotes how much time the processor dedicated to Process 2 has spent in executing the process. We assume that both processors start executing their processes at the same time. This is reflected in the first fact of Program 3.19, where T_1 and T_2 are constrained to be exactly 0. Intuitively, since we assume that time progresses uniformly everywhere[3], we ought to have that $T_1 = T_2$ everywhere in the semantics of the program. However, this is not the case with our CLP modeling. Informally, in our modeling T_1 is the end time of last executed statement of Process 1, with the condition that that statement has been executed at the point T_1' in time where $T_1' \leq T_2$. The semantics of T_2 is the same for Process 2. Several subtler points are:

- Notice that in this semantics, it is never the case that the difference of T_1 and T_2 reaches infinity since no statement takes infinite time, and when $T_i > T_j$, it cannot be the case that the statement of Process i is executed resulting in a least model where $T_i + \varepsilon > T_j$.

- In our CLP model, each statement is effective instantaneously at the start of its execution, and then the processor only delays until the required time for the statement ends. We could have adopted alternative modelings where a statement is effective at the end of its duration, or where variables are read at the start or the duration, and updated at the end of the duration.

[2]There are often arguments in the literature on whether it is best to use real or discrete time domain. Without taking side, here our intention is simply demonstrate that CLP is powerful enough to model *hybrid systems*, which are those with both discrete and continuous components.

[3]This is not true in the physical world.

```
$p(0,0,T_1,T_2,A,X,Y,N)$ :- $T_1 = 0, T_2 = 0, A[0] = 0, A[1] = 1$.
$p(1,L_2,T_1',T_2,A,2,Y,N)$ :- $inc(T_1,T_2,T_1'), p(0,L_2,T_1,T_2,A,X,Y,N)$.
$p(2,L_2,T_1',T_2,A,X,Y,N)$ :-
   $inc(T_1,T_2,T_1'), X \leq N, p(1,L_2,T_1,T_2,A,X,Y,N)$.
$p(\Omega,L_2,T_1',T_2,A,X,Y,N)$ :-
   $inc(T_1,T_2,T_1'), X > N, p(1,L_2,T_1,T_2,A,X,Y,N)$.
$p(3,L_2,T_1',T_2,A',X,Y,N)$ :- $inc(T_1,T_2,T_1')$,
   $A' = \langle A,X,A[X-1]+A[X-2]\rangle, p(2,L_2,T_1,T_2,A,X,Y,N)$.
$p(1,L_2,T_1',T_2,A,X+2,Y,N)$ :- $inc(T_1,T_2,T_1'), p(3,L_2,T_1,T_2,A,X,Y,N)$.
$p(L_1,1,T_1,T_2',A,X,3,N)$ :- $inc(T_2,T_1,T_2'), p(L_1,0,T_1,T_2,A,X,Y,N)$.
$p(L_1,2,T_1,T_2',A,X,Y,N)$ :-
   $T_2 \leq T_1, T_2' = T_2+300, p(L_1,1,T_1,T_2,A,X,Y,N)$.
$p(L_1,3,T_1,T_2',A,X,Y,N)$ :-
   $inc(T_2,T_1,T_2'), Y \leq N, p(L_1,2,T_1,T_2,A,X,Y,N)$.
$p(L_1,\Omega,T_1,T_2',A,X,Y,N)$ :-
   $inc(T_2,T_1,T_2'), Y > N, p(L_1,2,T_1,T_2,A,X,Y,N)$.
$p(L_1,4,T_1,T_2',A',X,Y,N)$ :- $inc(T_2,T_1,T_21)$,
   $A' = \langle A,Y,A[Y-1]+A[Y-2]\rangle, p(L_1,3,T_1,T_2,A,X,Y,N)$.
$p(L_1,2,T_1,T_2',A,X,Y+2,N)$ :- $inc(T_2,T_1,T_2'), p(L_1,4,T_1,T_2,A,X,Y,N)$.

$inc(T_1,T_2,T_1')$ :- $T_1 \leq T_2, T_1+95 \leq T_1' \leq T_1+105$.
```

Program 3.19: Dangerous Parallel Fibonacci CLP Model

We note here that timing can be added to sequential programs as well.

As has been explained in Chapter 2, where A is an array, we use the notation $A[I]$ to denote the I-th element of A, and $\langle A,I,J\rangle$ to denote the array resulting from replacing its I-th element in A by J.

In this section we have presented a concurrent program without synchronization between the processes. We have also considered various ways of handling real-time synchronization, but since this topic is less relevant to this thesis, its discussion is relegated to Section A.1 of the appendix.

3.5 Hardware Constraints

Similar to the previous example, we seek here to model an internal component of a program's execution, which in this case is the timing characteristics due to computer hardware used to run the program.

In the previous section we have exemplified how we may introduce clock variables in a backward CLP model. The semantics of the clock variables there are intuitive: it is the amount of time which have lapsed since the start of execution of the program. The semantics, however,

```
⟨0⟩  $j := 1$
⟨1⟩  **while** $(j < 3)$ **do**
⟨2⟩      **if** $(a[j] > a[j+1])$**then**
⟨3⟩          $swap(a[j], a[j+1])$
         **end if**
⟨4⟩      $j := j+1$
     **end do**
```

Program 3.20: Bubbling Loop

is not very straightforward for a backward model: It is the amount of time which must have lapsed from the start of the execution of the program, if the execution of the program is to have the chance to reach the point of interest in some time α from the start of the execution. In our modeling, we do not have to provide the constant α : It can instead be represented as another clock variable T_f which value is unchanging, and is the same as the value of a clock variable T at the program point of interest.

The example program that we use here is the inner loop of the bubble sort algorithm, which we call "Bubbling," shown as Program 3.20.

Now suppose that Bubbling is run on a direct-mapped instruction cache architecture. Here, there is a fixed assignment of cache line to statements. We assume the architecture has 2 cache lines: line 0 and 1, with each line contains at most 2 instructions. For Bubbling, statements labeled with program points $\langle 0\rangle$, $\langle 2\rangle$ and $\langle 4\rangle$ are mapped to cache line 0, while $\langle 1\rangle$ and $\langle 3\rangle$ to cache line 1. A cache hit costs 1 time unit, while a miss costs 5 time units.

We implement these assumptions in our CLP model shown as Program 3.21. The variables K and K' represent the cache configuration: a pair of lists (one for each cache line), and each list contains at most two statements. Cache update operation is modeled by the predicate *update*. Without giving their definitions, we note that The predicates *in* and *notin* represent tests of inclusion and non-inclusion, respectively, of a statement in a cache line.

3.6 Timed Safety Automata

In this section we focus on modeling of timed safety automata (TSA) specification in CLP to demonstrate that CLP can be used not only to model programs, but also high-level specifications which represent transition systems. Whereas in Section 3.3 we have shown how to model asynchronous concurrency in CLP. TSA specifications discussed here may contain both asynchronous

```
$p(0,A,K,J,T,T_f)$ :- $K = [[],[]]$,
    $update(0,K,K',E), p(1,A,K',1,T+E,T_f)$.
$p(1,A,K,J,T,T_f)$ :- $J < 3, update(1,K,K',E), p(2,A,K',J,T+E,T_f)$.
$p(1,A,K,J,T,T_f)$ :- $J \geq 3, update(1,K,K',E), p(5,A,K',J,T+E,T_f)$.
$p(2,A,K,J,T,T_f)$ :- $A[J] > A[J+1]$,
    $update(2,K,K',E), p(3,A1,K',J,T+E,T_f)$.
$p(2,A,K,J,T,T_f)$ :- $A[J] \leq A[J+1]$,
    $update(2,K,K',E), p(4,A,K',J,T+E,T_f)$.
$p(3,A,K,J,T,T_f)$ :- $swap(A,J,J+1,A')$,
    $update(3,K,K',E), p(4,A',K',J,T+E,T_f)$.
$p(4,A,K,J,T,T_f)$ :- $update(4,K,K',E), p(1,A,K',J+1,T+E,T_f)$.
$p(5,A,K,J,T,T)$.

$update(I,[L_0,L_1],[L_0,L_1],1)$ :- $in(I,L_0)$.
$update(I,[L_0,L_1],[L_0,L_1],1)$ :- $in(I,L_1)$.
$update(I,[L_0,L_1],[L'_0,L_1],5)$ :- $notin(I,L_0), notin(I,L_1)$,
    $update\_line(L_0,I,L'_0)$.
$update(I,[L_0,L_1],[L_0,L'_1],5) : -notin(I,L_0), notin(I,L_1)$,
    $update\_line(L_1,I,L'_1)$.
$update\_line([],I,[I])$.% cache empty
$update\_line([H_1],I,[H_1,I])$.% partial
$update\_line([\_,H_2],I,[H_2,I])$.% cache full
```

Program 3.21: Bubbling Loop Forward CLP Model

as well as synchronous concurrency.

3.6.1 Timed Automata and Timed Safety Automata

Timed automata [2] is a class of ω-automata (automata over infinite words) with *timed* words[4]. The alphabet of a timed automata is a pair of transition label and the occurrence time of the transition. A timed *safety* automaton (*TSA*) [99, 51] is a timed automaton without the ω-acceptance condition. Hence by definition it is simply a transition system.

An ω-acceptance imposes some liveness to a timed automaton. For instance, Büchi automaton accepts only infinite strings that visit any accepting state infinite number of time. A problem when implementing a timed automaton as a real system is that, given a particular stage of computation, it is in general undecidable to compute the amount of time which has to pass before the next transition is taken, such that the run of the system satisfies the acceptance condition. Waiting for too long or too short at a particular point alter the valuation of some transition guard in the future such that the accepting state is never reached. ω-acceptance therefore distinguish current execution state of the system as those that can possibly satisfy the acceptance condition

[4]For an introduction to ω-automata, see [191].

and those that are not.

By removing the acceptance condition, we declare that any reachable state is part of the valid behavior of the system, without further qualification. A reachable state of the system is a result of accepting (any one of possibly infinite number of) finite strings. This implies that the properties that we would be able to reason about now belongs to the *safety* class of properties[5]. As noted by Henzinger et al., properties verifiable using timed safety automata include *reachability* (or *possibility*), *invariance*, and *time-bounded inevitability*, which is a special kind of invariance [99].

3.6.2 State Transition Systems

Before we formally define TSA, we need to first define a concept of *state transition system*, in which the notion of *valuation* is essential. A *valuation* is a mapping of a variable into a value in its domain. We extend this notion to a sequence of variables in the obvious way. When a valuation γ maps the variable x of the integer domain to the number 1, we write $\gamma(x) = 1$. In the context of transition systems, we usually call a valuation as a *state*.

Definiton 3.4 (State Transition System). A transition system is a triple $\langle \tilde{X}, \Theta, \mathcal{R} \rangle$, where $\tilde{X}$ is a sequence of variables, and Θ is a set of *initial states*, and $\mathcal{R}$ is a binary relation relating two valuations. We write $\gamma \longrightarrow \gamma'$ to denote that $\langle \gamma, \gamma' \rangle \in \mathcal{R}$.

The semantics of a state transition system is a set of reachable states, which includes Θ, and all other states related to Θ by one or more applications of relations in $\mathcal{R}$.

It is always possible to represent Θ as a disjunctive constraint $D(\tilde{X}) \equiv \bigvee_{i=1}^{m} D_i(\tilde{X})$, where $D(\gamma(\tilde{X}))$ holds if and only if $\gamma(\tilde{X}) \in \Theta$. Similarly, it is always possible to represent $\mathcal{R}$ as a disjunctive constraint $R(\tilde{X}, \tilde{X}') \equiv \bigvee_{i=1}^{n} R_i(\tilde{X}, \tilde{X}')$, where $R(\gamma(\tilde{X}), \gamma'(\tilde{X}'))$ holds if and only if $\gamma(\tilde{X}) \longrightarrow \gamma(\tilde{X}')$.

It is obvious that the semantics of a state transition system corresponds to the least model of

[5]For a formal distinction between safety and liveness properties, see the book of Schneider [178], as well as the paper of Bjørner et al. [19]. Bjørner et al. define safety to be all properties representable using temporal logic with past operators only. In the semantics of past operators, the notion of "past" is well-founded (finite): it always starts from the time 0. This is not the case with future operators.

a CLP program which contains the clauses

$$\begin{array}{lll} p(\tilde{X}) & \texttt{:-} & D_1(\tilde{X}). \\ \vdots & & \\ p(\tilde{X}) & \texttt{:-} & D_m(\tilde{X}). \\ p(\tilde{X}') & \texttt{:-} & R_1(\tilde{X},\tilde{X}'), p(\tilde{X}). \\ \vdots & & \\ p(\tilde{X}') & \texttt{:-} & R_n(\tilde{X},\tilde{X}'), p(\tilde{X}). \end{array}$$

3.6.3 CLP Semantics of TSA

We have shown how we translate a state transition system into a CLP program. We now define the structure of a timed automata, whose semantics is given by a state transition system. For the translation into a CLP program, we simply assume the discussion in the previous section.

We have provided a formalization of TSA in [108]. Here we provide a formalization which closely follows Bengtsson and Yi [14], including a CCS-style composition.

Definiton 3.5 (Timed Safety Automaton). A timed safety automaton is a structure $\langle \Sigma, \mathcal{Q}, q_0, \mathcal{C}, \mathcal{D}, \iota, \Delta, I \rangle$ where:

- Σ is the input alphabet of *actions*,
- $\mathcal{Q}$ is a finite set of *locations*,
- q_0 is the *initial location*,
- $\mathcal{C}$ is a finite set of *clock variables* that range over nonnegative real numbers ($\mathbb{R}_+$),
- $\mathcal{D}$ is a finite set of *discrete variables* that range over integers and arrays,
- ι is the *initial valuation* of the discrete variables,
- $\Delta \subseteq Q \times \Sigma \times Q \times \mathcal{B}(\mathcal{C}) \times \mathcal{B}(\mathcal{D}) \times 2^{\mathcal{C}} \times \mathcal{R}(\mathcal{D}, \mathcal{D}')$ is the *transition relation*, where
  - $\mathcal{B}(\mathcal{C})$ is the set of constraints on $\mathcal{C}$,
  - $\mathcal{B}(\mathcal{D})$ is the set of constraints on $\mathcal{D}$,
  - $\mathcal{R}(\mathcal{D}, \mathcal{D}')$ is the set of all relations between the discrete variables and its primed versions,

and,

- $I : Q \mapsto \mathcal{B}(\mathcal{C})$ is a mapping that associates a *location invariant* to every location.

The fact that a TSA has both continuous (real) and discrete (integer or array) components means that it can be used for modeling *hybrid* systems.

We shall typically denote elements of the sets Σ, Q, $\mathcal{C}$, $\mathcal{B}(\mathcal{C})$, $\mathcal{B}(\mathcal{D})$, $2^{\mathcal{C}}$, and $\mathcal{R}(\mathcal{D},\mathcal{D}')$ by the following, possibly subscripted or primed symbols: *s*, *q*, *c*, $\varphi^{\mathcal{C}}$, $\varphi^{\mathcal{D}}$, *r* and ρ respectively.

Note that there are three kinds of actions in Σ : *internal*, *input* and *output* actions. Input actions are always written with postfix "?," and output actions are always written with postfix "!" Internal actions are written using neither postfix. Input and output actions are used in the parallel composition of timed automata.

Note that other works, for example, [2, 14] provide TSA definitions that limit the language of clock constraints $\mathcal{B}(\mathcal{C})$ to constraints of the form $c \odot n$, or $c_1 \odot c_2$, where $\odot \in \{\leq, <, =, >, \geq\}$, and n is a nonnegative integer. This is due to to computability issues. We do not impose such restriction to the language of constraints that we use. As we will show later, for some problems, more liberal use of constraint language still preserves decidability.

Given a transition $(q, s, q', \varphi^{\mathcal{C}}, \varphi^{\mathcal{D}}, r, \rho)$, q represents the current location, s the action that triggers the current transition, q' is the next location, $\varphi^{\mathcal{C}}$ is a constraint over $\mathcal{C}$ that must hold when the transition occurs, similarly $\varphi^{\mathcal{D}}$ over $\mathcal{D}$, and $r \subseteq \mathcal{C}$ a set of clock variables to be reset (assigned to 0) during the transition, and ρ is the set of updates to the discrete variables.

Given a TSA, a *clock valuation* is a mapping from its set of clock variables $\mathcal{C}$ to positive real numbers. On the set of valuations we define the following partial order.

Given a set of clock variables $\mathcal{C} = \{c_1, \ldots, c_k\}$, we say that a clock valuation γ satisfies a clock constraint $\varphi^{\mathcal{C}}$, written $\gamma \in \varphi^{\mathcal{C}}$, if the result of the substitution $\varphi^{\mathcal{C}}[c_1/\gamma(c_1), \ldots, c_k/\gamma(c_k)]$ is a ground constraint that holds.

Given two valuations γ_1, γ_2, we write $\gamma_1 \leq \gamma_2$ if $\gamma_1(c) \leq \gamma_2(c)$ for every clock variable $c \in \mathcal{C}$.

For simplicity, we shall assume that the location invariants are convex, i.e., given a location invariant $I(q)$ at location q, and three clock valuations γ_1, γ_2 and γ_3 such that $\gamma_1 \leq \gamma_2 \leq \gamma_3$, we have that $\gamma_2 \in I(q)$ whenever $\gamma_1 \in I(q)$ and $\gamma_3 \in I(q)$.

We denote by $\gamma + d$, where $d \in \mathbb{R}_+$ the clock valuation γ' where for all clock variable $c \in \mathcal{C}$, $\gamma'(c) = \gamma(c) + d$.

Definiton 3.6 (Operational Semantics of a Timed Safety Automaton). TSA is a transition system where states are triples $\langle q,\gamma,\delta\rangle$, with the set Θ of initial states contains all valuations $\langle q_0,\gamma,\delta\rangle$, satisfying $\gamma \in I(q_0)$, and $\delta \in \iota$. The state transition system has the following two kinds of of transitions:

- *Delay transition* $\langle q,\gamma,\delta\rangle \longrightarrow \langle q,\gamma+d,\delta\rangle$ when $\gamma \in I(q)$ and $(\gamma+d) \in I(q)$ for $d \in \mathbb{R}_+$. Note that here, all clock valuations are incremented by the *same amount d*.
- *Discrete transition* $\langle q,\gamma,\delta\rangle \longrightarrow \langle q',\gamma',\delta'\rangle$ when $(q,s,q',\varphi^{\mathcal{C}},\varphi^{\mathcal{D}},r,\rho) \in \Delta$, $\gamma \in \varphi^{\mathcal{C}}$, $\delta \in \varphi^{\mathcal{D}}$, $\gamma' = \gamma[r/0]$, $\gamma' \in I(q')$ and $\rho(\delta,\delta')$.

So far we have introduced the semantics of a timed safety automaton (following [14]). As a system specification, however, we more often use a number of automata rather than a single automaton. Here we provide the semantics of the parallel composition of a number of automata, also following [14].

We are given a set of TSAs $\{T_1,\dots,T_n\}$, where each T_k is the structure $\langle \Sigma_k, Q_k, q_0^k, \mathcal{C}_k, \mathcal{D}_k, \iota_k, \Delta_k, I_k\rangle$, for all k, $1 \le k \le n$. The *parallel composition* of $T_1,\dots,T_n$ is a transition system where the state is $\langle \tilde{q},\tilde{\gamma},\tilde{\delta}\rangle$, where $\tilde{q}$ is a vector $(q_1,\dots,q_n)$, $q_i \in Q_i$, and $\tilde{\gamma}$ is a clock valuation of the set of clocks $\bigcup_{i=1}^n \mathcal{C}_i$, and similarly $\tilde{\delta}$ is a valuation of the discrete variables in $\bigcup_{i=1}^n \mathcal{D}_i$. We write $\tilde{\gamma} \in I(\tilde{q})$ in place of $\tilde{\gamma} \in \bigwedge_{i=1}^n I(q_i)$.

Definiton 3.7 (Semantics of Parallel Timed Safety Automata). The parallel composition of TSA is a transition system which has $\langle (q_0^1,\dots,q_0^n),\tilde{\gamma},\tilde{\delta}\rangle$ in the set of initial states, where $\tilde{\gamma} \in I((q_0^1,\dots,q_0^n))$, and $\tilde{\delta} \in \iota_1 \wedge \dots \wedge \iota_n$, and with transitions of the following three types:

- *Delay transitions* which advance the time, but without executing an action:

  $$\langle \tilde{q},\tilde{\gamma},\tilde{\delta}\rangle \longrightarrow \langle \tilde{q},\tilde{\gamma}+d,\tilde{\delta}\rangle,$$

  when $\tilde{\gamma} \in I(\tilde{q})$ and $\tilde{\gamma}+d \in I(\tilde{q})$.
- *Discrete internal transitions* represent a transition that executes an internal action in one of the automaton T_i :

  $$\langle \tilde{q},\tilde{\gamma},\tilde{\delta}\rangle \longrightarrow \langle \tilde{q}[q_i' \mapsto q_i],\tilde{\gamma}',\tilde{\delta}'\rangle,$$

  when $(q_i,s,q_i',\varphi^{\mathcal{C}},\varphi^{\mathcal{D}},r,\rho) \in \Delta_i, \tilde{\gamma} \in \varphi^{\mathcal{C}}, \tilde{\delta} \in \varphi^{\mathcal{D}}, \tilde{\gamma}' = \tilde{\gamma}[r \mapsto 0]$, $\rho(\tilde{\delta},\tilde{\delta}')$ and $\tilde{\gamma}' \in I(\tilde{q}[q_i' \mapsto q_i])$.

- *Discrete synchronization transitions* represent the simultaneous execution of an input action $s?$ of automaton T_i and an output action $s!$ of automaton T_j :

  $$\langle \tilde{q}, \tilde{\gamma}, \tilde{\delta} \rangle \longrightarrow \langle \tilde{q}[q_i' \mapsto q_i][q_j \mapsto q_j'], \tilde{\gamma}', \tilde{\delta}' \rangle,$$

  when $(q_i, s?, q_i', \varphi_i^{\mathcal{C}}, \varphi_i^{\mathcal{D}}, r_i, \rho_i) \in \Delta_i, (q_j, s!, q_j', \varphi_j^{\mathcal{C}}, \varphi_j^{\mathcal{D}}, r_j, \rho_j) \in \Delta_j, i \neq j, \tilde{\gamma} \in \varphi_i^{\mathcal{C}} \wedge \varphi_j^{\mathcal{C}}, \tilde{\delta} \in \varphi_i^{\mathcal{D}} \wedge \varphi_j^{\mathcal{D}}, \tilde{\gamma}' = \tilde{\gamma}[r_i \cup r_j \mapsto 0], \rho_i(\tilde{\delta}, \tilde{\delta}'), \rho_j(\tilde{\delta}, \tilde{\delta}')$ and $\tilde{\gamma}' \in I(\tilde{q}[q_i' \mapsto q_i][q_j' \mapsto q_j])$.

Denote by T the parallel composition of the timed safety automata $T_1, \ldots, T_n$. Intuitively, each T_i runs independently inside T. Since the alphabets Σ_i, $1 \leq i \leq n$ are not necessarily disjoint, by the 3rd transition above, it is often the case that an event triggers synchronous transitions in two parallel TSA, where one of the TSA executes an output action, and the other execute the input version of the same action; such synchronous transitions are the means by which the TSAs communicate with one another.

Figure 3.2 shows a graphical representation of a train crossing system specified as the parallel composition of three TSAs: each representing the train, the controller and the gate. It is the TSA version of the timed automata example in [2]. When the train approaches the gate, a sensor emits an "approach" signal, denoted by the output action **approach!**. This signal is detected by the controller, modeled using the transition with the input signal **approach?** in the controller automaton. After at least 2 time units approaching the crossing, the train enters the crossing, expressed by the transition with the internal action **in**. After some time the train exits the crossing, executing the internal action **out**. A sensor, which detects the train exiting the crossing then executes an **exit!** output action which again is synchronized with the **exit?** input action of the controller. All these must occur in less than 5 t.u. since the train approaches the crossing, hence the constraint $c < 5$ given on the transition.

The function of the controller is to receive signals from the train and send appropriate instructions to the gate. When a train approach is detected by the controller, it then executes a **lower!** output action which is synchronized with its input version in the gate. This transition has to be executed at exactly 1 t.u. since the approach of the train was detected. Similarly, when the controller detects that the train has exited the crossing, it then instructs the gate to raise using the **raise!** action. This must happen less than 1 t.u.

The gate simply receives instructions from the controller to lower and raise the gate. There are constraints on time taken for the gate to be fully lowered (when the action **down** is executed),

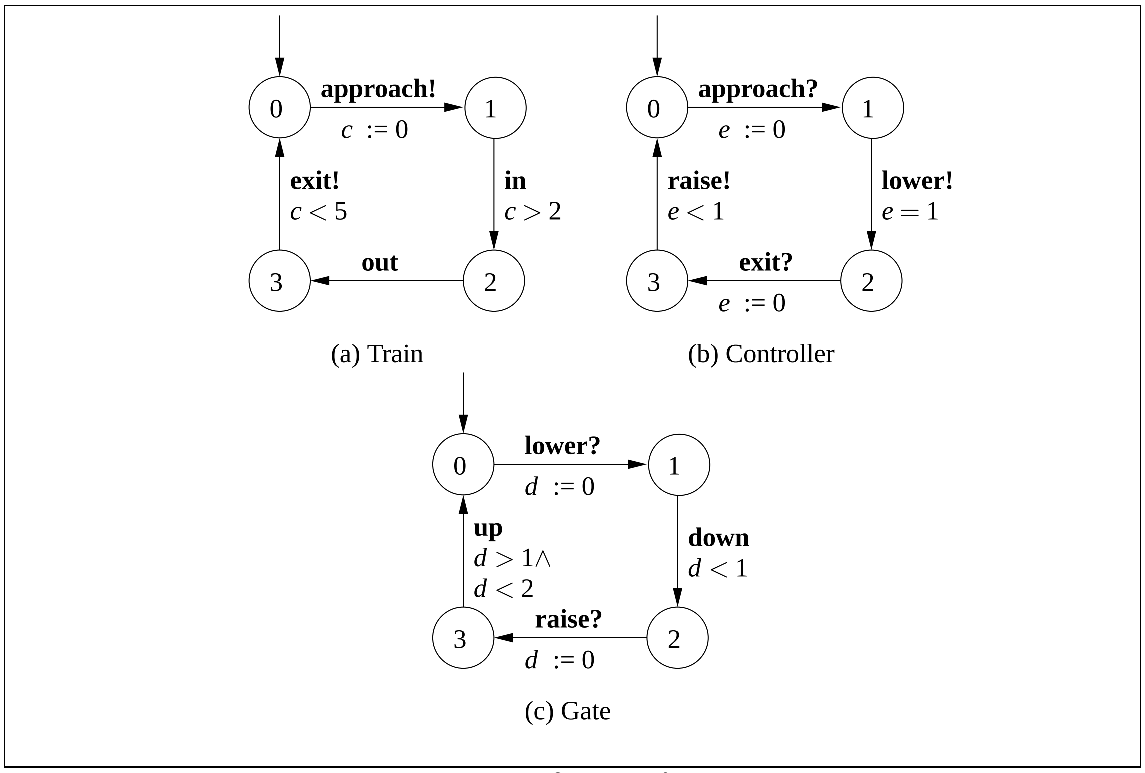

Figure 3.2: TSA Specification of a Train Crossing

and for the gate to be fully raised (the execution of the action **up**).

Figure 3.3 shows the TSA representing the parallel composition of the three TSAs in Figure 3.2. We note that the events **approach** and **exit** are in the alphabets of both the train and the controller, while the events **lower** and **raise** are in the alphabets of both the gate and the controller. All these four symbols trigger synchronous transitions; for example, the transition from location $[1,0,1]$ to location $[1,1,2]$ on event **lower** in Figure 3.3 represents synchronous transitions of both the gate and the controller, which in Figure 3.2 is represented both as the output and input action **lower!** and **lower?**.

We can define the semantics of TSA as the set of reachable states that can be reached from the initial state by a sequence of either delay or discrete transitions. In the sequence, it does not matter how many times delay transitions is continually taken, since any number of delay transitions can always be replaced by a single delay transition which delays the same amount of time. Even nonexistent delay transition between two discrete transitions can be represented by a single delay transition whose delay amount is 0. Therefore we may assume that TSA only has one kind of transition which consists of taking a discrete transition and immediately taking a delay to advance the clocks. Of course, the delay must satisfy the location invariant of the target location.

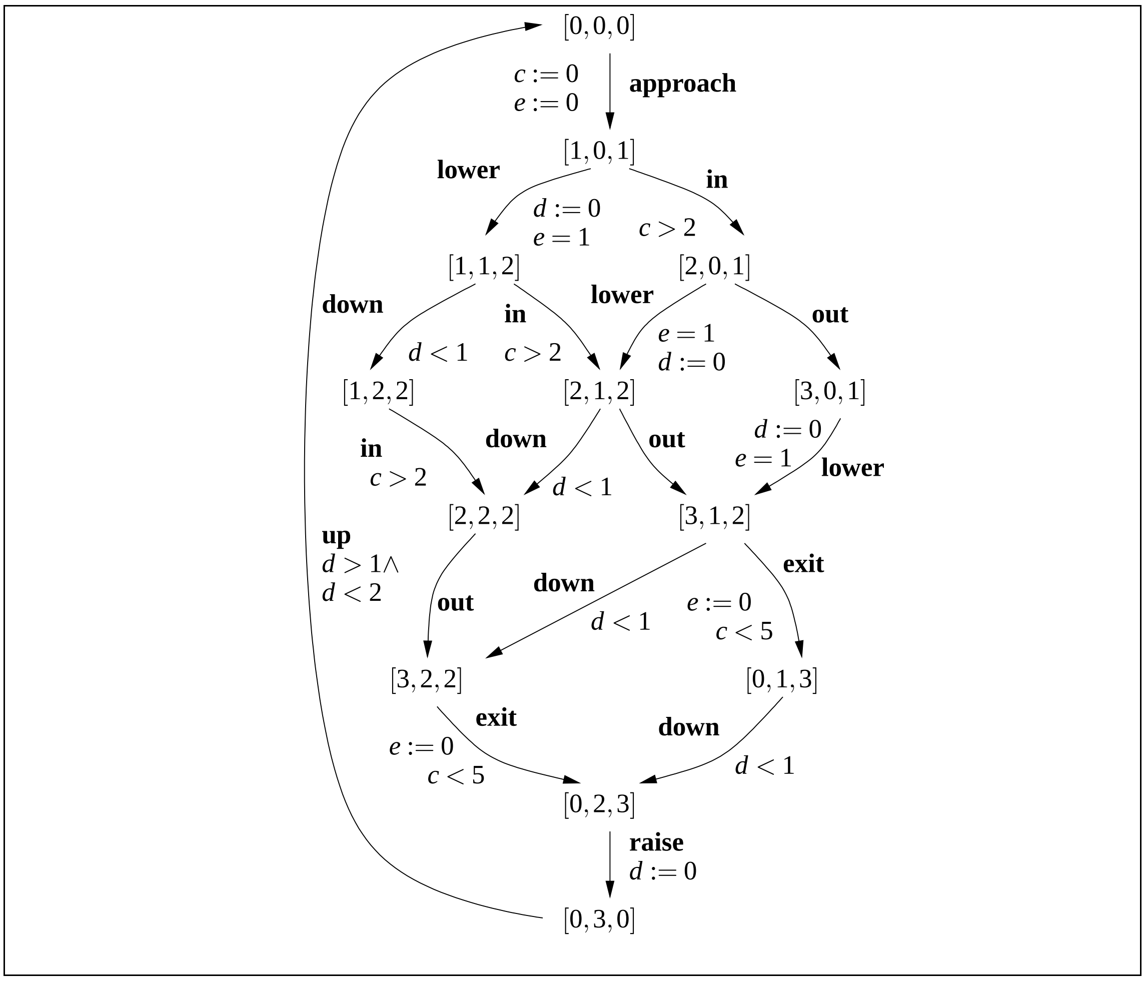

Figure 3.3: TSA Parallel Composition

This allows us to reduce the transition relations given in Definition 3.7 into just the following two classes:

- *Internal transitions* represents a transition that executes an internal action in one of the automaton T_i, immediately followed by some delay $d \geq 0$:

$$\langle \overline{q}, v \rangle \longrightarrow \langle \overline{q}[q_i' \mapsto q_i], v' \rangle,$$

  when $(q_i, s, q_i', B, r) \in \Delta_i, v \in B, v' = v[r \mapsto 0] + d$ and $v' \in I(\overline{q}[q_i' \mapsto q_i])$.

- *Synchronization transitions* represent the simultaneous execution of an input action $s?$ of automaton T_i and an output action $s!$ of automaton T_j, immediately followed by some delay $d \geq 0$:

$$\langle \overline{q}, v \rangle \longrightarrow \langle \overline{q}[q_i' \mapsto q_i][q_j \mapsto q_j'], v' \rangle,$$

  when $(q_i, s?, q_i', B_i, r_i) \in \Delta_i, (q_j, s!, q_j', B_j, r_j) \in \Delta_j, i \neq j, v \in B_i \wedge B_j, v' = v[r_i \cup r_j \mapsto 0] + d,$

and $v' \in I(\overline{q}[q'_i \mapsto q_i][q'_j \mapsto q_j])$.

As we have shown in Section 3.6.2, the translation of a state transition system into CLP program clauses is straightforward.

Let us now look again at our train crossing example. Note that in the train crossing example, there is no discrete variables, nor state invariants. The initial states of the parallel composition is described by the following constraint fact:

$$p(0,0,0,C,D,E) \texttt{ :- } C \geq 0, C = D = E.$$

Since the initial locations have no location invariant, the clock valuations need only satisfy that they belong to $\mathbb{R}_+$. The clock variables must have the same value, since they can only by incremented (by a delay transition) by the same amount of time.

The CLP program clauses modeling internal transitions are the followings:

$$\begin{aligned}
&p(2,P_2,P_3,C+Delta,D+Delta,E+Delta) \texttt{ :- } \\
&\quad p(1,P_2,P_3,C,D,E), C > 2, Delta \geq 0. \\
&p(3,P_2,P_3,C+Delta,D+Delta,E+Delta) \texttt{ :- } \\
&\quad p(2,P_2,P_3,C,D,E), Delta \geq 0. \\
&p(P_1,P_2,2,C+Delta,D+Delta,E+Delta) \texttt{ :- } \\
&\quad p(P_1,P_2,1,C,D,E), D < 1, Delta \geq 0. \\
&p(P_1,P_2,0,C+Delta,D+Delta,E+Delta) \texttt{ :- } \\
&\quad p(P_1,P_2,3,C,D,E), D > 1, D < 2, Delta \geq 0.
\end{aligned}$$

Note that P_1, P_2 and P_3 are variables representing the locations of the train, controller and gate, respectively. Also, all clocks in the above transitions are incremented by the same nonnegative *Delta*.

The CLP program clauses modeling synchronization transitions are the followings:

$$
\begin{aligned}
&p(1,1,P_3,Delta,D+Delta,Delta)\ \ \text{:-}\\
&\quad p(0,0,P_3,C,D,E),Delta \geq 0.\\
&p(0,3,P_3,C+Delta,D+Delta,Delta)\ \ \text{:-}\\
&\quad p(3,2,P_3,C,D,E),C<5,Delta \geq 0.\\
&p(P_1,2,1,C+Delta,Delta,E+Delta)\ \ \text{:-}\\
&\quad p(P_1,1,0,C,D,E),E=1,Delta \geq 0.\\
&p(P_1,0,3,C+Delta,Delta,E+Delta)\ \ \text{:-}\\
&\quad p(P_1,3,2,C,D,E),E<1,Delta \geq 0.
\end{aligned}
$$

The possible transitions of the train crossing can actually be represented graphically as in Figure 3.3. Notice that each arrow is a specialization of one of the CLP clauses above.

3.6.4 More Examples

As we have already noted above, we do not restrict the clock constraints on locations and transition guards to be of certain form. CLP modeling of timed automata is even more flexible: it can be used to model timed automaton not according to the standard definition given above.

Example 3.6. (Worker TSA) Consider a timed automaton in Figure 3.4 describing a daily schedule of a worker. The worker starts from home each day, staying in office for at most 10 hours, during which some of the time is spent in the cafe. The worker may visit the cafe a number of times, and we want to record the total amount of time he spends in the cafe. For this purpose we introduce the variable Y which is not a clock, but represents a real number variable. In a transition, a non-clock variable is never incremented uniformly, as with the clock variables. Notice that in Figure 3.4, we freely assign the clock values X and Z to the variable Y, and use the variable together with a clock variable X in a transition guard. This is not allowed in our formalization of TSA given above. We provide the CLP model of the timed automata as Program 3.22.

Existing timed automata analysis tools are limited in their expressiveness due to the model checking algorithm used. This is because the standard algorithms for analysis of timed automata depend on an analysis of clock regions [14], which use a more restrictive class of real constraints. Not only we can enlarge the class of allowed specifications, we can actually automatically verify

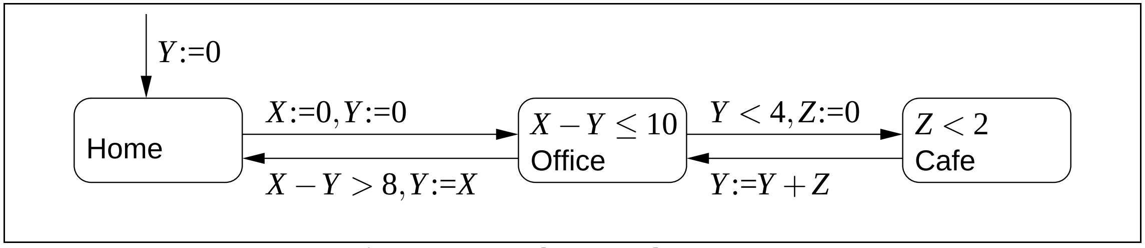

Figure 3.4: Worker Timed Automaton

$p(0,X,Y,Z)$:- $X \geq 0, X = Z, Y = 0.$
$p(1,X',Y',Z')$:- $Delta \geq 0, X' = Delta, Z' = Z + Delta,$
  $Y' = 0, X' - Y \leq 10, p(0,\_,Y,Z).$
$p(2,X',Y',Z')$:- $Delta \geq 0, X' = X + Delta, Y' = Y, Z' = Delta,$
  $Y < 4, X - Y \leq 10, Z' < 2, p(1,X,Y,\_).$
$p(1,X',Y',Z')$:- $Delta \geq 0, X' = X + Delta, Y' = Y + Z, Z' = Z + Delta,$
  $Z < 2, X' - Y' \leq 10, p(2,X,Y,Z).$
$p(0,X',Y',Z')$:- $Delta \geq 0, X' = X + Delta, Y' = X, Z' = Z + Delta,$
  $8 < X - Y, X - Y \leq 10, p(1,X,Y,Z).$

Program 3.22: Worker CLP Model

the example, which we will demonstrate in later chapters.

Example 3.7. (Fischer's Algorithm TSA) For this experiment we used the standard Fischer's algorithm; see Figure 3.5 where *i* ranges over the number of processes. Location 3 is the critical section of each process.

The working of the TSA can be explained informally as follows. There is a maximum delay 2 time units for the process to stay at location 1, and there is a minimum delay 4 time units for a process to stay at location 2. Suppose that there are two processes that are to enter their critical sections (location 3). Both processes must have taken at most 2 time units at location 1 and set K to its own process id. Since both processes also wait at location 2 for more than 4 time units, both processes must have finished their attempt to set K to its own process id. At state 2, both processes thus can judge correctly based on the value of K whether to enter its critical section (location 3) or not (back to location 0).

We show the backward CLP model of two-automata Fischer's algorithm TSA as Program 3.23.

Example 3.8. (Bridge Crossing Problem) Here we consider a modification of the bridge crossing problem from UPPAAL 3.4.6 package. The system consists of 2 kinds of automata: a controller (Figure 3.6) and a number of trains (Figure 3.7). When N is the number of trains, our

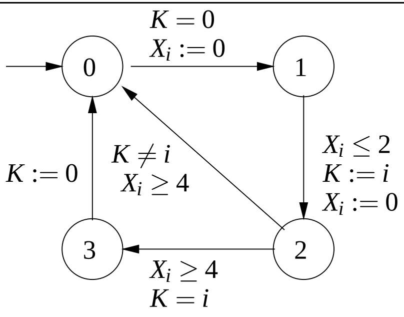

Figure 3.5: Fischer's Algorithm TSA for Process *i*

$$
\begin{array}{l}
p(0,0,X_1,X_2,K) \text{ :- } K = 0, Z \geq 0, X_1 = Z, X_2 = Z.\\
p(1,L_2,Z,X_2+Z,K) \text{ :- } p(0,L_2,X_1,X_2,K), K = 0, Z \geq 0.\\
p(2,L_2,Z,X_2+Z,K) \text{ :- } p(1,L_2,X_1,X_2,K'), K = 1, X_1 \leq 2, Z \geq 0.\\
p(3,L_2,X_1+Z,X_2+Z,K) \text{ :- } p(2,L_2,X_1,X_2,K), K = 1, X_1 \geq 4.\\
p(0,L_2,X_1+Z,X_2+Z,K) \text{ :- } p(3,L_2,X_1,X_2,K'), K = 0, Z \geq 0.\\
p(0,L_2,X_1+Z,X_2+Z,K) \text{ :- } p(2,L_2,X_1,X_2,K), K \neq 1, X_1 \geq 4, Z \geq 0.\\
p(L_1,1,X_1+Z,Z,K) \text{ :- } p(L_1,0,X_1,X_2,K), K = 0, Z \geq 0.\\
p(L_1,2,X_1+Z,Z,K) \text{ :- } p(L_1,1,X_1,X_2,K'), K = 2, X_2 \leq 2, Z \geq 0.\\
p(L_1,3,X_1+Z,X_2+Z,K) \text{ :- } p(L_1,2,X_1,X_2,K), K = 2, X_2 \geq 4, Z \geq 0.\\
p(L_1,0,X_1+Z,X_2+Z,K) \text{ :- } p(L_1,3,X_1,X_2,K'), K = 0, Z \geq 0.\\
p(L_1,0,X_1+Z,X_2+Z,K) \text{ :- } p(L_1,2,X_1,X_2,K), K \neq 2, X_2 \geq 4, Z \geq 0.
\end{array}
$$

Program 3.23: Two-Process Fischer's Algorithm TSA Backward CLP Model

modeling uses $2N+1$ variables, where $X_1,\ldots,X_N$ are clocks for each train, the next N variables $Pos_0,\ldots,Pos_N$ represent the positions of each train in a global queue, and the last variable *Len* denotes the number of trains in the queue.

The original UPPAAL model contains *committed locations*, which are locations where no time progress is allowed. In our modeling, we translate the sequence of transitions that visit committed locations between both endpoints into single transition. This does not change the semantics since committed locations are not part of UPPAAL state space [13]. Also, in the original model, trains that are to enter the crossing are kept in a queue. Instead of implementing the queue, as mentioned above, we use the variables Pos_i to model the position of each train *i* in the queue. Our modeling allows for simple modeling of symmetry property later (Section 4.5.2).

We show the CLP model of two-trains bridge crossing problem TSA as Program 3.24.

Example 3.9. **(Dining Philosophers with Timeout)** Here we verify a real-time solution to the dining philosophers problem whose automaton is shown in Figure 3.8. We assume there are N philosophers, where $N \geq 3$. Each philosopher can be in any of 3 states: *thinking* (at location 0), *hungry* (at location 1), and *eating* (at location 2). Location 0 is the initial state of every

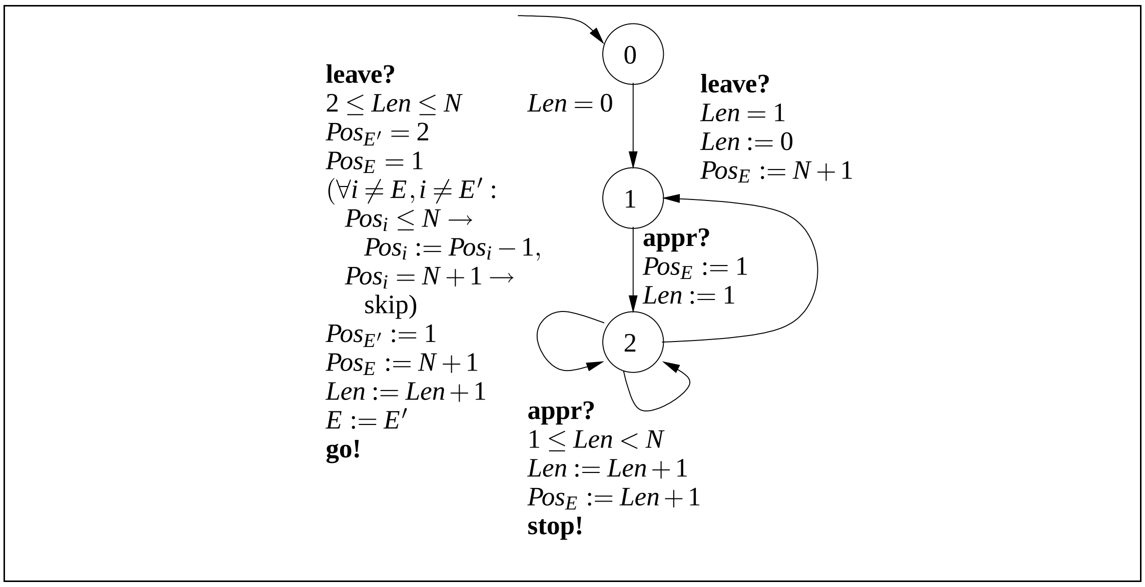

Figure 3.6: Bridge Crossing Controller TSA

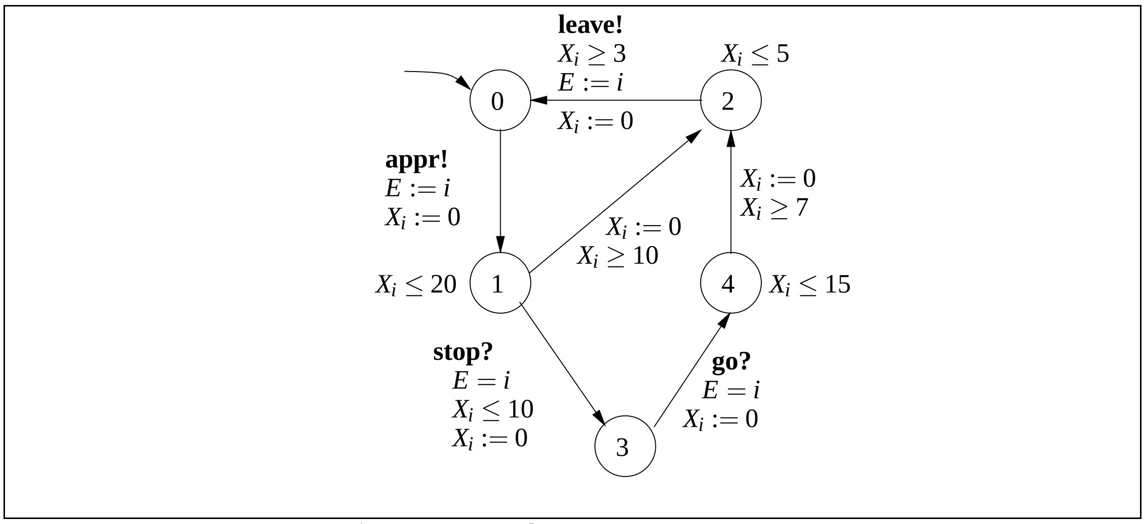

Figure 3.7: Bridge Crossing Train TSA

philosopher. The location may change from 0 to 1 when philosopher i picks a fork i, denoted by changing the value of variable F_i from 0 to 1 (there are N forks in the system, modeled as variables F_1 to F_N). Its location changes from 1 to 2 when philosopher i picks the fork $F_{next(i)}$, where $next(i) = (i \textbf{ mod } N) + 1$. From location 2, a philosopher may return to location 0 by setting both F_i and $F_{next(i)}$ to 0. To avoid deadlock, a philosopher may return to location 0 from location 1 if it cannot continue to location 2 in less than 2 time units. We show the CLP model for three dining philosophers as Program 3.25.

```
$p(0,0,0,X_1,X_2,Pos_1,Pos_2,Len)$ :- $Z \geq 0, X_1 = Z, X_2 = Z, Pos_1 = 3, Pos_2 = 3.$
$p(1,L_1,L_2,X_1+Z,X_2+Z,Pos_1,Pos_2,0)$ :- $p(0,L_1,L_2,X_1,X_2,Pos_1,Pos_2,Len).$
$p(1,0,L_2,Z,X_2+Z,3,3,0)$ :- $p(2,2,L_2,X_1,X_2,1,3,1),$
    $X_1 \geq 3, X_1 \leq 5, Z \geq 0.$
$p(1,L_1,0,X_1+Z,Z,3,3,0)$ :- $p(2,L_1,2,X_1,X_2,3,1,1),$
    $X_2 \geq 3, X_2 \leq 5, Z \geq 0.$
$p(2,0,4,Z,Z,3,1,1)$ :- $p(2,2,3,X_1,X_2,1,2,2),$
    $X_1 \geq 3, X_1 \leq 5, X_2 \leq 15, Z \geq 0.$
$p(2,4,0,Z,Z,1,3,1)$ :- $p(2,3,2,X_1,X_2,2,1,2),$
    $X_2 \geq 3, X_2 \leq 5, X_1 \leq 15, Z \geq 0.$
$p(2,3,L_2,Z,X_2+Z,Len+1,Pos_2,Len+1)$ :- $p(2,0,L_2,X_1,X_2,3,Pos_2,Len),$
    $Len \geq 0, Len \leq 1, Z \geq 0.$
$p(2,L_1,3,X_1+Z,Z,Pos_1,Len+1,Len+1)$ :- $p(2,L_1,0,X_1,X_2,Pos_1,3,Len),$
    $Len \geq 0, Len \leq 1, Z \geq 0.$
$p(2,1,L_2,Z,X_2+Z,1,Pos_2,1)$ :- $p(1,0,L_2,X_1,X_2,3,Pos_2,0),$
    $Z \leq 20, Z \geq 0.$
$p(2,L_1,1,X_1+Z,Z,Pos_1,1,1)$ :- $p(1,L_1,0,X_1,X_2,Pos_1,3,0),$
    $Z \leq 20, Z \geq 0.$
$p(L_0,2,L_2,Z,X_2+Z,Pos_1,Pos_2,Len)$ :- $p(L_0,1,L_2,X_1,X_2,Pos_1,Pos_2,Len),$
    $X_1 \leq 20, X_1 \geq 10, Z \leq 5, Z \geq 0.$
$p(L_0,L_1,2,X_1+Z,Z,Pos_1,Pos_2,Len)$ :- $p(L_0,L_1,1,X_1,X_2,Pos_1,Pos_2,Len),$
    $X_2 \leq 20, X_2 \geq 10, Z \leq 5, Z \geq 0.$
$p(L_0,2,L_2,Z,X_2+Z,Pos_1,Pos_2,Len)$ :- $p(L_0,4,L_2,X_1,X_2,Pos_1,Pos_2,Len),$
    $X_1 \leq 7, X_1 \geq 15, Z \leq 5, Z \geq 0.$
$p(L_0,L_1,2,X_1+Z,Z,Pos_1,Pos_2,Len)$ :- $p(L_0,L_1,4,X_1,X_2,Pos_1,Pos_2,Len),$
    $X_2 \leq 7, X_2 \geq 15, Z \leq 5, Z \geq 0.$
```

Program 3.24: Two-Trains Bridge Crossing Backward CLP Model

3.7 Statecharts

Statechart is a popular modeling language, which is part of UML. It was originally introduced by Harel [88], as a variant of *hypergraph* [89]. In this section we show how we *compositionally* model in CLP a Statechart train crossing example in [12], which specification is given in Figure 3.9. Our exposition here will be rather informal.

Overview of various semantics of Statechart is given by von der Beeck [195] but excluding STATEMATE semantics. STATEMATE semantics is described by Harel and Naamad [92, 91], and Harel and Politi [93]. Some comparisons of Statemate and UML semantics of statechart have also been provided by Eshuis et al. [63]. With respect to compositionality, the same also discussed by Simons [181]. Bhaduri and Ramesh provide a survey on various approaches to model checking of Statechart [16].

Among the compositional approaches to Statechart verification, Alur and Yannakakis' approach is based on identification of repeating superstate templates [3], while Damm et al.'s is

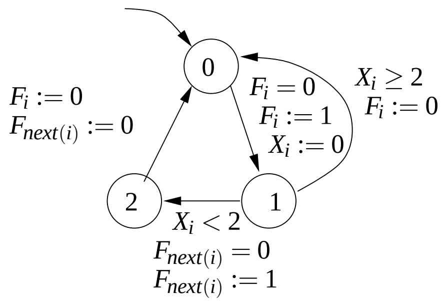

Figure 3.8: Real-Time Dining Philosophers

$p(0,0,0,0,0,0,Z,Z,Z)$:- $Z \geq 0$.
$p(1,L_2,L_3,1,F_2,F_3,Z,X_2+Z,X_3+Z)$:- $p(0,L_2,L_3,0,F_2,F_3,X_1,X_2,X_3),Z \geq 0$.
$p(2,L_2,L_3,F_1,1,F_3,X_1+Z,X_2+Z,X_3+Z)$:- $p(1,L_2,L_3,F_1,0,F_3,X_1,X_2,X_3),X_1 < 2,Z \geq 0$.
$p(0,L_2,L_3,0,0,F_3,X_1+Z,X_2+Z,X_3+Z)$:- $p(2,L_2,L_3,F_1,F_2,F_3,X_1,X_2,X_3),Z \geq 0$.
$p(0,L_2,L_3,0,F_2,F_3,X_1+Z,X_2+Z,X_3+Z)$:- $p(1,L_2,L_3,F_1,F_2,F_3,X_1,X_2,X_3),X_1 \geq 2,Z \geq 0$.
$p(L_1,1,L_3,F_1,1,F_3,X_1+Z,Z,X_3+Z)$:- $p(L_1,0,L_3,F_1,0,F_3,X_1,X_2,X_3),Z \geq 0$.
$p(L_1,2,L_3,F_1,F_2,1,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,1,L_3,F_1,F_2,0,X_1,X_2,X_3),X_2 < 2,Z \geq 0$.
$p(L_1,0,L_3,F_1,0,0,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,2,L_3,F_1,F_2,F_3,X_1,X_2,X_3),Z \geq 0$.
$p(L_1,0,L_3,F_1,0,F_3,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,1,L_3,F_1,F_2,F_3,X_1,X_2,X_3),X_2 \geq 2,Z \geq 0$.
$p(L_1,L_2,1,F_1,F_2,1,X_1+Z,X_2+Z,Z)$:- $p(L_1,L_2,0,F_1,F_2,0,X_1,X_2,X_3),Z \geq 0$.
$p(L_1,L_2,2,1,F_2,F_3,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,L_2,1,0,F_2,F_3,X_1,X_2,X_3),X_3 < 2,Z \geq 0$.
$p(L_1,L_2,0,0,F_2,0,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,L_2,2,F_1,F_2,F_3,X_1,X_2,X_3),Z \geq 0$.
$p(L_1,L_2,0,F_1,F_2,0,X_1+Z,X_2+Z,X_3+Z)$:- $p(L_1,L_2,1,F_1,F_2,F_3,X_1,X_2,X_3),X_3 \geq 2,Z \geq 0$.

Program 3.25: Three-Process Real-Time Dining Philosophers CLP Model

based on interface computation and abstraction (over-approximation) by the underlying model checker [39, 17, 18], and Behrmann et al.'s approach is based on under-approximation [12]. Some works provide compositional semantics of Statecharts, such as [44, 194, 79, 131].

In our discussion on TSA above we have introduced the notion of locations. In Statechart argot, a location is called a *state*. However, to avoid confusion, we shall use the term "location" instead of "state." A Statechart specification often only has locations, although it is conceptually easy to introduce variables and their guards similar to clocks and guards of TSA. For example, the work of David et al. is on translating models in a Statechart variant called *Timed Hierarchical Automata*, which may be augmented with clocks, into timed (safety) automaton [42]. Similarly, Eshuis et al. also allows augmentation of statecharts with real-number clock variables [63].

An important feature of a Statechart specification is that it is hierarchical, in which a location may contain one or more parallel Statecharts. For example, in Figure 3.9, the top-level location Root contains the parallel statecharts Train and Crossing, while the location Move contains one statechart, which is also named Move. We call a location containing more than one parallel

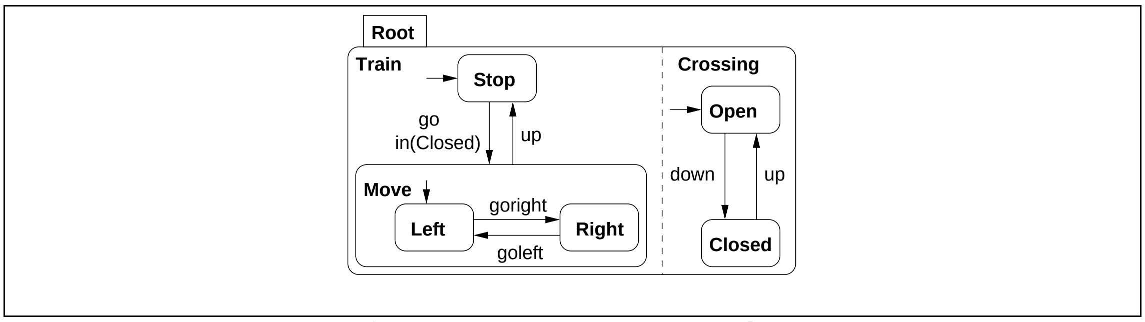

Figure 3.9: Train Crossing Statechart

statecharts as an *AND location*, while we call a location containing one statechart as an *OR location*. Hence, the top-level location Root is an AND location, while the location Move is an OR location. We call a location not containing other statecharts as *primitive* location. In Figure 3.9, Stop, Left, Right, Open, and Close are all primitive locations.

As is a TSA, a statechart is also a state transition system. The state of a statechart is called a *configuration* [92]. A configuration specifies the current active location of each statechart in a specification.

A statechart specifies an *initial configuration* and transitions from a configuration to another by the triggering of an event. Here we assume that events are generated by the environment. The events in Figure 3.9 are goright, goleft, go, up, and down. Common Statechart variants allow events to be generated by the transitions themselves[6]

We represent a configuration as a term of the syntax $s(Name, Subconfiguration\_List)$, where *Name* is the name of the location, and *Subconfiguration_List* is the list of configurations of the parallel sub-statecharts included in the location *Name*. At any time, the configuration of a primitive location has an empty subconfiguration list, those for an OR state has only one element in its subconfiguration list, while the configuration of an AND state has more than one. For example, the initial configuration of the statechart in Figure 3.9 is

$$s(root, [s(train, [s(stop, [])]), s(crossing, [s(closed, [])])]).$$

The subconfiguration list is dynamically changing, depending on the currently active locations.

We show as Program 3.26 our backward CLP model of the statechart in Figure 3.9. We model the events using numbers, where goright=0, goleft=1, go=2, up=3, and down=4. Notice that we separate individual statecharts into different predicate. The predicates *rootinit*and

[6]This rises the issues of *step* versus *superstep* semantics. The former only executes all the transitions at the occurrence of an event, while the latter also executes subsequently generated internal events until stable state is reached. We are allowed to ignore this issue here.

*roottrans*encode the initial configuration and the transitions of the Statechart Root. Similarly with *traininit*and *traintrans*which encode the initial configuration and transitions of the Statechart Train, and so on. The arguments of the *trans* predicate of each statecharts (*roottrans*, *traintrans*, *crossingtrans*, and *movetrans*) require some explanation. Among the arguments, the second holds the current event, while the third and fourth hold the configuration before and after the transition of the statechart, respectively. The first argument holds the topmost configuration before the transition, which in this example is the configuration of Root. This is because there can be guards such as in(Closed) which requires us to refer to the topmost configuration (which always include other configurations) to check for transition enabledness.

The configuration space of the train crossing example is given by the interpretation of the *p* predicate in the least model of Program 3.26.

In modeling, we had been ignoring the issue of transition priority. Here, an event may both trigger a transition in the higher and lower level of hierarchy. Statemate semantics prioritizes higher-level transitions [92, 91, 93], while UML semantics prioritizes lower-level transitions [150]. Although we could have adopted any of them, we have opted not to for the sake of cleaner modeling. The modeling is also flexible enough to be extended with "history states" found in some Statechart models.

Since a statechart is hierarchical, it is only natural to independently verify statecharts of different hierarchy, or of different parallel components. Thus an important requirement of a Statechart verification system is its *compositionality*. Some approaches to statechart verification have some kind of compositionality [38, 39, 3, 17, 18, 12, 129, 177]. Those that are not compositional are based on translating the top-level statechart into a flat state transition system based on a given semantics (e.g., Statemate or UML), for example [78, 142, 123, 42, 128, 126, 163, 63]. As can be seen in Program 3.26, our modeling is obviously compositional, where each statechart is given separate predicate. Reasoning independently about a subchart can be done by adding a predicate which initial configuration is defined by the *init* predicate, and the transitions are defined by the *trans* predicate corresponding to the subchart. For example, for independently reasoning on the Move subchart, we add the predicate *q* as follows:

$$
\begin{aligned}
q(C) &\;\texttt{:-}\; moveinit(C).\\
q(C') &\;\texttt{:-}\; movetrans(\_,0,C,C'),q(C).\\
q(C') &\;\texttt{:-}\; movetrans(\_,1,C,C'),q(C).
\end{aligned}
$$

```
p(C') :- roottrans(C,0,C,C'), p(C).
p(C') :- roottrans(C,1,C,C'), p(C).
p(C') :- roottrans(C,2,C,C'), p(C).
p(C') :- roottrans(C,3,C,C'), p(C).
p(C') :- roottrans(C,4,C,C'), p(C).
p(C) :- rootinit(C).

rootinit(s(root,[s(train,[C1]),s(crossing,[C2])])) :-
   traininit(C1), crossinginit(C2).
roottrans(C,E,s(root,[s(train,[C1]),s(crossing,[C2])]),
   s(root,[s(train,[C1']),s(crossing,[C2'])])) :-
   traintrans(C,E,C1,C1'), crossingtrans(C,E,C2,C2').

traininit(s(stop,[])).
traintrans(C,2,s(stop,[]),s(move,[C1'])) :-
   moveinit(C1'), in(C,s(closed,_)).
traintrans(C,3,s(move,_),s(stop,[])).
traintrans(C,E,s(move,[C1]),s(move,[C1'])) :-
   movetrans(C,E,C1,C1').

crossinginit(s(open,[])).
crossingtrans(C,3,s(closed,[]),s(open,[])).
crossingtrans(C,4,s(open,[]),s(closed,[])).
crossingtrans(C,E,C1,C1) :- E ≤ 2.

moveinit(s(left,[])).
movetrans(C,0,s(left,[]),s(right,[])).
movetrans(C,1,s(right,[]),s(left,[])).
movetrans(C,E,C1,C1) :- E ≥ 2.
```

Program 3.26: Train Crossing CLP Model

The set of reachable configurations of the subchart Move is now given by the interpretation of q in the least model.

Full compositional verification with Statecharts, however is not easy. Compositionality favors Statemate semantics instead of UML semantics, since with Statemate semantics, the reachability higher in hierarchy cannot be canceled by the behavior of a subchart [181]. There are also other issues such as transition guards which may refer to location of arbitrary subcharts. This makes us unable to translate the reachability question of a parallel component into a global setting.

Chapter 4

Correctness Specifications

To reason about constraint logic programs, we need a way of specifying properties that we want to prove on them. This chapter is devoted into introducing assertions for this purpose.

4.1 Assertions

We use assertions of the syntax

$$G \models H,$$

where both G and H are goals possibly containing CLP program predicates. G and H may refer to a set of common variables, say $\tilde{X}$. We denote by $\tilde{Y}$ the variables that only occur in G, and by $\tilde{Z}$ the variables that only occur in H. The above assertion has the following logical semantics:

$$\langle \forall \tilde{X}, \tilde{Y} : G \Rightarrow \langle \exists \tilde{Z} : H \rangle \rangle.$$

For clarity, in an assertion $G \models H$, the variables in $\tilde{Z}$ will be prefixed with "?"[1].

The properties that we can specify using assertions belong to the safety class. In the remainder of this chapter we demonstrate how we may specify an extensive class of safety properties, not only simple invariance property, but also invariance on pointer data structures, and even *structural* or *non-behavioral* properties, such as symmetry of programs. In the next section we first start with invariance, which we call traditional safety.

[1]We attribute this notational convention to Fribourg [74].

4.2 Traditional Safety

The notion of invariance is well-known in the literature, e.g. in [144]. It states the condition (constraint) that all states of a program must satisfy. In stating traditional safety, H contains no CLP program predicate. Here we proceed by example.

Example 4.1. When the interpretation of p is given by Program 3.2, the following assertion states that at the end of execution of Program 3.1, the relation $s = (n^2 - n)$ holds:

$$p(\Omega, X, S, N) \models S = (N^2 - N)/2.$$

Example 4.2. An obvious property that we would like to verify on the bakery algorithm (Program 3.14) is that it guarantees mutual exclusion, the property which can be represented using the assertion

$$p(2, 2, X, Y) \models \Box,$$

where p is interpreted in the least model of Program 3.15. The above assertion states that it is impossible for the both processes to be in their critical section (program point $\langle 2 \rangle$) at the same time. More technically, none of $p(L_1, L_2, X, Y)$ where $L_1 = 2$ and $L_2 = 2$ is included in the interpretation of p in Program 3.15. Alternatively, to specify the same property, one can also write

$$p(L_1, L_2, X, Y) \models L_1 \neq 2 \vee L_2 \neq 2.$$

Example 4.3. When the interpretation of p is given by Program 3.17, the following assertion states that at any state of Program 3.16, the relation $x \leq 3y$ always holds:

$$p(L_1, L_2, Q, X, Y) \models X \leq 3Y.$$

Example 4.4. In Program 3.18, when $n \leq 3$, both processes never access the same array location. That is, at the end of the execution we can guarantee that $a[i] = \mathit{fib}(i)$, where $\mathit{fib}(i)$ denotes i-th Fibonacci number. This property can be represented by the assertion

$$p(\Omega, \Omega, T_1, T_2, A, X, Y, N), N \leq 3 \models A[N] = \mathit{fib}(N),$$

interpreted on Program 3.19. For $n > 3$, computing $a[i]$ could precede computing of $a[i-1]$ for some i, that is, the correctness of Program 3.18 is not guaranteed in case $n > 3$.

Example 4.5. To state that the execution time bound of Program 3.20 is 30 time units, we use the assertion

$$p(0, A, K, J, T, T_f) \models T_f - T \leq 30,$$

whose predicate p is interpreted by the CLP Program 3.21.

Example 4.6. To specify on TSA in Figure 3.4 that the worker is never more than 20 hours outside the house on his work day, we use the assertion

$$p(0, X, Y, Z) \models Y \leq 20,$$

whose predicate p is interpreted using the CLP Program 3.22.

Example 4.7. On the statechart of Figure 3.9, a property “the train is not in the state move while the crossing in the state open” can be specified using the assertion

$$p(C), in(C, s(move, \_)), in(C, s(open, \_)) \models \Box,$$

where p is interpreted in the least model of CLP Program 3.26.

4.3 Array Safety

We now start with an example to show how we may specify traditional safety properties which are constructed using CLP predicates. Here we use an assertion to state the correctness property of the bubble sort algorithm (Program 3.5) given in Section 3.1.5. A suitable correctness condition would be that at the end of the execution of the program, the array is sorted. To state this property,

$$\begin{array}{lll} sorted(A,I,N) & :- & I = N-1. \\ sorted(A,I,N) & :- & I < N-1, A[I] \leq A[I+1], sorted(A,I+1,N). \end{array}$$

Program 4.1: *Sorted*

$$\begin{array}{lll} allz(H,X,X) & :- & H[X] = 0, X \neq 0. \\ allz(H,X,Y) & :- & allz(H,X,T), H[Y] = 0, H[T+1] = Y, Y \neq 0. \end{array}$$

Program 4.2: Nonempty All-Zero Linked List I

we first need to define what we mean by a sorted array. This we define as the CLP Program 4.1. $sorted(a,i,n)$ holds for any value a, i, and n if and only if a is an array, where its elements with indices from i to n are sorted.

The correctness of Program 3.5 now can be stated as the assertion

$$p(0,A,I,J,N,A_f,N_f), I = 0 \models sorted(A_f,0,N_f). \tag{4.1}$$

4.4 Recursive Data Structures

CLP programs can be used, not only for providing semantics to programs, but also to specify recursive data structures properties. This is an extension to the correctness specification of program with array in the previous section. Here we view the heap as an array and specify the correctness of the heap using CLP predicates.

Recall the example programs in Section 3.1.6 which manipulate the heap. Let us discuss Program 3.7 which resets the elements of a list to 0. The correctness statement that we may want to ensure here is that at the end of the execution of the program, the elements of the list has their values set to 0. In order to state this property, we need a definition of a linked list whose elements are all 0. Here we show how we may construct one.

Using the modeling of program heap using array H as in Section 3.1.6, we may define recursive data structures using CLP programs. We can model a nonempty linked list whose all of its elements have the value 0 using Program 4.2. The least model of the program is represented by the following set:

$$
\begin{aligned}
\{allz(h,x,y) \quad | \quad & x \neq 0 \wedge h[x] = 0 \wedge \\
& h[x+1] \neq 0 \wedge h[h[x+1]] = 0 \wedge \\
& h[h[x+1]+1] \neq 0 \wedge h[h[h[x+1]+1]] = 0 \wedge \\
& \qquad \qquad \qquad \qquad \ldots \\
& h[\ldots h[x+1]\ldots] \neq 0 \wedge h[h[\ldots h[x+1]\ldots]] = 0 \wedge \\
& y = h[\ldots h[x+1]\ldots]\}.
\end{aligned}
$$

That is, in the least model, the first argument of *allz* can only be a heap containing a linked list with all zero elements starting from the pointer x up to the pointer y.

The correctness statement of Program 3.7 can now be specified as:

$$p(0,H,P,H_f,P_f), P \neq 0 \models allz(H_f,P,?Last), H[?Last+1] = P_f, P_f = 0.$$

That is, if we start the execution of Program 3.7, with input a non-empty list, then at the end of execution we will obtain a non-empty list with all of its elements reset to 0, and the *next* member of the last element is 0 (null). Note here that the variable *Last* only appears at the lhs of the assertion and is existentially quantified.

Although our specification looks good enough at first, we may need a more precise specification in order to prove stronger property later. Program 4.2, for example, does not specify more precisely how such linked list with all zero elements are constructed. Using array update expressions, we may specify that the linked list is constructed from an original heap, already containing a linked list, by assigning all elements of the original linked list to 0. We show the CLP program as Program 4.3. The *allz* predicate now has an extra last argument which is a placeholder of the updated heap.

Notice that Program 4.3 "traverses" the list from the head to the last element. We can also define a reverse of this, which traverses the list from the last element to the head of the list, as shown as Program 4.4.

Using either Program 4.3 or 4.4, we can re-state the correctness property of the linked list reset program more precisely as:

$$p(0,H,P,H_f,P_f), P \neq 0 \models allz(H,P,?Last,H_f), H[?Last+1] = P_f, P_f = 0. \qquad (4.2)$$

$$\begin{array}{lcl} allz(H,X,X,\langle H,X,0\rangle) & :- & X \neq 0. \\ allz(H,X,Y,\langle H_1,X,0\rangle) & :- & allz(H,T,Y,H_1), H[X+1]=T, X \neq 0. \end{array}$$

Program 4.3: Nonempty All-Zero Linked List II

$$\begin{array}{lcl} allz(H,X,X,\langle H,X,0\rangle) & :- & X \neq 0. \\ allz(H,X,Y,\langle H_1,Y,0\rangle) & :- & allz(H,X,T,H_1), H[T+1]=Y, X \neq 0. \end{array}$$

Program 4.4: Nonempty All-Zero Linked List III

Now we discuss another example: A linked list reverse program originally appeared in [166], which is Program 4.5. The forward CLP model is Program 4.6. (Note that we perform simplification by removing the CLP representation of the variable k, which only appears locally in the sequence of assignments in the loop body.)

The correctness statement for this program is that at the end of execution, we obtain a list which is a reverse of the original. For this purpose, we need to specify using a CLP program, what it means for two linked lists to be a reverse of each other. First note that what we roughly mean as reverse here is that whenever $x{\rightarrow}next$ points to y in the first linked list, $y{\rightarrow}next$ points to x in the second linked list. Our first attempt at modeling this in CLP is Program 4.7. Our informal interpretation of *reverse* here is that $reverse(h,i,j,h_1,i_1)$ holds when in heap h, a linked list segment from i up to but not including j has a 0-terminating reverse in another heap h_1, starting from the address i_1, where in h, $i_1 \rightarrow next = j$ (i_1 is the address of the node immediately pointing to j in the original list).

Program 4.7, however, does not specify that the two heaps h and h_1 are related. Hence, it says nothing on whether the data values are preserved or not. Instead, we may want to specify a stronger property that the second heap is an update of the first heap (without changing the data values), for which we can use a second version of *reverse* given as Program 4.8.

For this problem we want to prove, that given an acyclic list with head I, when the program finishes, we obtain an acyclic list with head J, which is the reverse of the original list. We express this property as the following assertion:

$$p(0,H,I,J,H_f,J_f), alist(H,I), J=0 \models reverse(H,I,0,H_f,J_f), alist(H_f,J_f). \qquad (4.3)$$

In the assertion, we use the predicate *alist* (Program 4.9) to specify that the program is given

```
Initially $i \neq 0, j = 0$.
$\langle 0 \rangle$  **while** $(i \neq 0)$ **do**
$\langle 1 \rangle$      $k$ := $i{\rightarrow}next$
         $i{\rightarrow}next$ := $j$
         $j$ := $i$
         $i$ := $k$
     **end do**
```

Program 4.5: Linked List Reverse

```
$p(0,H,I,J,H_f,J_f)$ :- $p(1,H,I,J,H_f,J_f), I \neq 0$.
$p(0,H,I,J,H_f,J_f)$ :- $p(\Omega,H,I,J,H_f,J_f), I = 0$.
$p(1,H,I,J,H_f,J_f)$ :- $p(0,\langle H,I+1,J\rangle,H[I+1],I,H_f,J_f)$.
$p(\Omega,H,I,J,H,J)$.
```

Program 4.6: Linked List Reverse CLP Model

input a null-terminating acyclic linked list. Note that *alist* definition includes a call to *no_reach* (Program 4.10). *No_reach*(h,x,y) states that the heap h contains a null-terminating linked list which starts from address y, without any node stored at address x. The inclusion of *no_reach* will be important for the proof of the assertion which is to be given in Chapter 5. We have previously provided another definition of *no_reach* in Chapter 3 (Program 3.11) to specify that a pointer is not shared by a binary tree instead of linked list.

Note that because of the definition of *reverse* used (Program 4.8), the assertion (4.3) also implies

1. an “in-situ” property of the list reversal, where the memory region occupied by the list is unchanged,
2. that whenever node A points to B in the input, and B is not 0, node B points to node A in the output, and
3. that the program leaves unchanged memory regions outside the input list.

Both no. 1 and 2 above are guaranteed by the fact that Program 4.8 specifies that updates are done only on the memory region occupied by the list and nowhere else.

Now let us revisit our binary search tree insertion program given in Section 3.1.6 (Program 3.9 and its CLP model Program 3.10). We may want to prove that, given a binary search tree, and a value as an input, at the end of the execution of the routine, we obtain a binary search tree where the value to be inserted initially is included. To express this, we design a predicate *bst*,

$$\begin{array}{l} reverse(H,I,I,H_1,0). \\ reverse(H,I,Y,H_1,T) \ \text{:-}\ \ H[T+1]=Y, H_1[T+1]=J, reverse(H,I,T,H_1,J). \end{array}$$

Program 4.7: First Version of *Reverse/5*

$$\begin{array}{l} reverse(H,I,I,H,0). \\ reverse(H,I,Y,\langle H_1,T+1,J\rangle,T) \ \text{:-}\ \ H[T+1]=Y, reverse(H,I,T,H_1,J). \end{array}$$

Program 4.8: Second Version of *Reverse/5*

where $bst(h,x,min,max)$ holds when in the heap h the pointer x is a root of a binary search tree whose minimum value is *min* and the maximum value is *max*. Now, using the predicate *bst* we write the correctness statement of our binary search tree routine as follows:

$$\begin{gathered} p(0,H,X,A,H_f,X_f), X_0 = X, X \neq 0, bst(H,X_0,Min,Max) \\ \models bst(H_f,X_0,min(A,Min),max(A,Max)). \end{gathered} \tag{4.4}$$

In the above, *min* and *max* are functions that return the minimum and maximum of two numbers, respectively.

We define the predicate *bst* as Program 4.11. In the program, we also use the a call to *no_reach* similar to Program 4.9. Here we use our definition of *no_reach* for binary tree which is Program 4.12, copied here from Program 3.11 for convenience. Here, $no\_reach(h,x,y)$ means that the heap h contains a null-terminating binary tree which is rooted at address y, without any of its node stored at address x. The definition of *bst* contains calls to *no_share* (Program 4.13), a predicate which is essential for the proof.

Next, using the example of Rugina [173], we introduce a re-balancing routine of an AVL tree after node insertion, shown as Program 4.14. The routine is given an input an unbalanced subtree rooted at x, where its left subtree is two deeper than its right subtree, and at its left child, the left subtree is 1 deeper than its right subtree. As the output, we expect to obtain a balanced AVL tree. A CLP model of the above program is the Program 4.15.

We define using CLP Program 4.16 the specification of a balanced AVL tree. Intuitively, $avltree(h,x,d)$ holds if and only if h is heap containing an AVL tree rooted at x, and whose depth

$$\begin{aligned} alist(H,L) &\ \texttt{:-}\ L = 0. \\ alist(H,L) &\ \texttt{:-}\ L \neq 0, alist(H, H[L+1]), no\_reach(H, L, H[L+1]). \end{aligned}$$

Program 4.9: *Alist*

$$\begin{aligned} no\_reach(H,I,L) &\ \texttt{:-}\ L = 0. \\ no\_reach(H,I,L) &\ \texttt{:-}\ L \neq 0, I \neq L, \\ &\quad no\_reach(H, I, H[L+1]). \end{aligned}$$

Program 4.10: *No_reach* for Linked Lists

is d. Now, the correctness of the AVL tree can be stated via the following assertion.

$$\begin{aligned} & p(0,H,X,Y,Z,H_f,Y_f), avltree(H, H[X+2], DL-2), \\ & \quad avltree(H, H[H[X+1]+1], DL-1), avltree(H, H[H[X+1]+2], DL-2), \\ & \quad no\_share(H, X, H[X+2], H[H[X+1]+1], H[H[X+2]+2]) \\ & \qquad \models avltree(H_f, Y_f, DL). \end{aligned} \tag{4.5}$$

Here we include a *no_share* atom (defined as Program 4.13) directly in the assertion. Again, this is important for the proof.

4.5 Relative Safety

Relative safety declares that whatever invariant property holds for a subset of reachable states of the program also holds for another subset. More simply put, a state is reachable if another is. Note that this does not mean that the two states share a computation path.

Relative safety can be used to specify structural properties of programs, such as symmetry. Symmetry has been widely used as a state-space reduction technique in model checking, for instance, in Murφ [107] and SMC [183] among many others. Since symmetry induces an equivalence relation between program states, efficiency in state exploration can be achieved by only checking the representatives of the equivalence classes. Symmetry reduction in our proof method will be discussed in Section 5.6.

We briefly repeat our discussion in Chapter 1 that the concept of relative safety more powerful in handling symmetry than other approaches. The flexibility is gained from the fact that relative safety assertions specify relations on the reachable states only, whereas other approaches such as [183] requires the symmetry of the computation tree for the purpose of temporal logic

```
bst(H,X,H[X],H[X]) :- H[X+1] = 0, H[X+2] = 0.
bst(H,X,MinL,MaxR) :-
   H[X+1] ≠ 0, H[X] > MaxL, bst(H,H[X+1],MinL,MaxL),
   H[X+2] ≠ 0, H[X] < MinR, bst(H,H[X+2],MinR,MaxR),
   no_reach(H,X,H[X+1]),
   no_reach(H,X,H[X+2]),
   no_share(H,H[X+1],H[X+2]).
bst(H,X,MinL,H[X]) :-
   H[X+1] ≠ 0, H[X] > MaxL, bst(H,H[X+1],MinL,MaxL),
   H[X+2] = 0, no_reach(H,X,H[X+1]).
bst(H,X,H[X],MaxR) :-
   H[X+1] = 0, no_reach(H,X,H[X+2]),
   H[X+2] ≠ 0, H[X] < MinR, bst(H,H[X+2],MinR,MaxR).
```

Program 4.11: *Bst*

```
no_reach(H,I,L) :- L = 0.
no_reach(H,I,L) :- L ≠ 0, I ≠ L,
   no_reach(H,I,H[L+1]),
   no_reach(H,I,H[L+2]).
```

Program 4.12: *No_reach* for Binary Tree

verification, and hence more restrictive.

For simplicity here we restrict our discussion to backward model of programs in CLP, therefore we speak about reachable states (from the initial states), and not the set of states from which the state of interest is reachable. For detailed discussion on this issue, we refer the reader to Chapter 3. As we have discussed there, in this case the set of reachable states of a program is represented by the interpretation in the least model, of a certain predicate p of the CLP program. Given a CLP program defining a predicate p, relative safety says that a restriction Ψ_2 on the interpretation of the predicate p is in the least model if a restriction Ψ_1 is, written:

$$p(\tilde{X}_1), \Psi_1 \models p(\tilde{X}_2), \Psi_2.$$

Without loss of generality, we assume that the variable sequences $\tilde{X}_1$ and $\tilde{X}_2$ are disjoint, and the constraint Ψ_1 only refers to variables in $\tilde{X}_1$, while the constraint Ψ_2 may refer to both the variables in $\tilde{X}_1$ and $\tilde{X}_2$.

Other than symmetry, relative safety assertions can also be used to specify commutativity and serializability, which we will exemplify in the next section. Symmetry, commutativity, and

```
$no\_share(H, L_1, L_2)$ :- $L_1 = 0$.
$no\_share(H, L_1, L_2)$ :- $L_1 \neq 0$,
    $no\_reach(H, L_1, L_2)$,
    $no\_share(H, H[L_1 + 1], L_2)$,
    $no\_share(H, H[L_1 + 2], L_2)$.
```

Program 4.13: *No_share* for Binary Tree

```
⟨0⟩  $y$ := $x \rightarrow left$
⟨1⟩  if ($y \rightarrow val = 1$) then
⟨2⟩      $x \rightarrow val$ := 0
⟨3⟩      $y \rightarrow val$ := 0
⟨4⟩      $z$ := $y \rightarrow right$
⟨5⟩      $y \rightarrow right$ := $x$
⟨6⟩      $x \rightarrow left$ := $z$
     end if
```

Program 4.14: AVL Tree Rebalancing Routine

serializability are examples of structural or non-behavioral properties, i.e., properties determined by the structure of the program, and which are not necessarily related to the intended result of the computation. Relative safety is potentially useful to specify many other useful non-behavioral properties, possibly ad-hoc and application specific. The class of such properties is potentially large. It is intuitively clear that such information can help in speeding up the proof process of other properties, which we will demonstrate in later chapters.

Example 4.8. For example, suppose that we have a program whose variables are x_1 and x_2. The semantics of the program is given by a CLP program which defines the predicate p in the manner of Chapter 3. Now, the relative safety assertion $p(X_1, X_2) \models p(Y_1, Y_2), X_1 = Y_2, X_2 = Y_1$ (or more succinctly, $p(X_1, X_2) \models p(X_2, X_1)$) asserts that if the state $x_1 = \mu_1, x_2 = \mu_2$ is reachable, then so is $x_1 = \mu_2, x_2 = \mu_1$, for all values μ_1 and μ_2. In other words, the observable values of the two program variables x_1 and x_2 commute.

Example 4.9. (Permutational Symmetry) As another example, let us re-visit the two-process bakery algorithm (Program 3.14). Now consider a relative safety assertion, stating symmetry for the program as follows:

$$p(L_1, L_2, X, Y) \models p(L_2, L_1, Y, X). \tag{4.6}$$

$$
\begin{array}{lll}
p(0,H,X,Y,Z,H_f,Y_f) & :- & p(1,H,X,H[X+1],Z,H_f,Y_f). \\
p(1,H,X,Y,Z,H_f,Y_f) & :- & H[Y]=1, p(2,H,X,Y,Z,H_f,Y_f). \\
p(2,H,X,Y,Z,H_f,Y_f) & :- & p(3,\langle H,X,0\rangle,X,Y,Z,H_f,Y_f). \\
p(3,H,X,Y,Z,H_f,Y_f) & :- & p(4,\langle H,Y,0\rangle,X,Y,Z,H_f,Y_f). \\
p(4,H,X,Y,Z,H_f,Y_f) & :- & p(5,H,X,Y,H[Y+2],H_f,Y_f). \\
p(5,H,X,Y,Z,H_f,Y_f) & :- & p(6,\langle H,Y+2,X\rangle,X,Y,Z,H_f,Y_f). \\
p(6,H,X,Y,Z,H_f,Y_f) & :- & p(\Omega,\langle H,X+1,Z\rangle,X,Y,Z,H_f,Y_f). \\
p(\Omega,H,X,Y,Z,H,Y). & &
\end{array}
$$

Program 4.15: AVL Tree Rebalancing Routine CLP Model

$$
\begin{array}{l}
\mathit{avltree}(H,0,0). \\
\mathit{avltree}(H,X,D_1+1) \ :- \\
\quad H[X]=D_1-D_2, 0 \le D_1-D_2, D_1-D_2 \le 1, \\
\quad \mathit{avltree}(H,H[X+1],D_1), \mathit{avltree}(H,H[X+2],D_2). \\
\mathit{avltree}(H,X,D_2+1) \ :- \\
\quad H[X]=D_1-D_2, D_1-D_2=-1, \\
\quad \mathit{avltree}(H,H[X+1],D_1), \mathit{avltree}(H,H[X+2],D_2).
\end{array}
$$

Program 4.16: *Avltree*

We may also imagine a three-process bakery algorithm, for which the following two assertions are sufficient specify to all of the possible symmetries on the reachable states:

$$p(L_1,L_2,L_3,X,Y,Z) \models p(L_2,L_1,L_3,Y,X,Z).$$

$$p(L_1,L_2,L_3,X,Y,Z) \models p(L_1,L_3,L_2,X,Z,Y).$$

The first assertion specifies the transposition of Process 1 and 2, while the second assertion specifies those between Process 2 and 3. Other permutations are achievable by some composition of the above 2 transpositions. This example shows how our relative safety assertion handles what is known in the literature as *permutational* symmetry, for example, as defined and used in [107].

4.5.1 Group-Theoretical Symmetry

Here we clarify that the symmetry properties as defined by relative safety assertions are indeed the symmetry known in mathematics, for example, see [199].

Symmetry is a group of transformations that preserves the transformed object, that is, it is an *automorphism*. Automorphisms in geometrical objects include translation, reflection and rotation. For example, let us have a look at a frieze shown in Figure 4.1. Let us assume that the frieze extends to infinity. In this situation, one transformation that preserves the frieze is

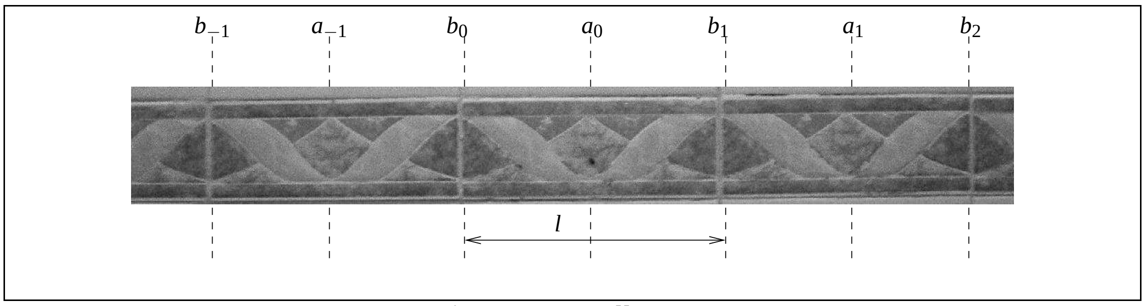

Figure 4.1: Wall Frieze

translation of length l to the left (or right), and also translation of length nl to the left (or right), where n is a natural number or 0. When $n = 0$, the transformation is an *identity*. Another transformation that preserves the frieze is reflection on the axes that cut through the middle of a unit pattern, indicated with the letter a_i in Figure 4.1, where i is an integer or the line separating two unit patterns, indicated with b_i. The only rotation possible here is 360° rotation around any point on the two-dimensional plane, which is just the identity. The transformation can also be composed of several transformations, for example, reflection on axis a_0 followed by translation of length l to the right is also a transformation.

Any automorphism must be included in a group with the composition of automorphisms as its operator [199]. Such a group is known as an *automorphism group*. Given an object, we may determine its set of all transformations that are automorphisms on it. The composition of multiple transformations is also an automorphism. Moreover, a set of all automorphisms on a given object induces a group Γ, with the following characteristics:

- The identity transformation belongs to Γ.
- If automorphism s belongs to Γ, then so is its inverse s^{-1}.
- If s and t belong to Γ, then so does their composite $s \cdot t$.

Let us define $Aut_\Gamma(x,y)$ if and only if there exists an automorphism $s \in \Gamma$ such that x is transformed to y. It is easy to see that the relation Aut_Γ is an equivalence.

As in many other works on symmetry for verification, in this thesis we will be interested with model-theoretic instead of geometric automorphisms. However, such automorphism also has geometric consequences, hence researchers have been concerned with automorphisms on a possibly infinite computation tree, for example in [183]. Our work differs from the rest where we are only concerned about automorphisms that preserve (do not restrict or enlarge) the set of reachable states. This is enough to obtain reduction in safety verification, while allowing us to specify symmetry in more cases.

Our idea is to use a set of relative safety assertions to specify possible automorphisms on reachable states, which then induces an equivalence relation on them. Here, a single relative safety assertion in general only describes a partial mapping, while an automorphism is total. In general we need a set of assertions to describe a total mapping, say π. Moreover, equivalence between states is obtained by also proving a complete set of assertions which represent all the mappings in an automorphism group. This would include inverses, whose proof is often straightforward. Suppose that $map(G \models H)$ is the mapping represented by the relative safety assertion $G \models H$. Now, as an example, the symmetry assertion (4.6) for the two-process bakery algorithm characterizes an automorphism group *Aut* on the set of reachable states as follows:

- We include the obvious $map(p(L_1,L_2,X,Y) \models p(L_1,L_2,X,Y))$ in *Aut* satisfying the existence of identity.
- By simple renaming $\{L_1 \mapsto L_2, L_2 \mapsto L_1, X \mapsto Y, Y \mapsto X\}$ on assertion (4.6), the reverse $map(p(L_2,L_1,Y,X) \models p(L_1,L_2,X,Y))$ is in *Aut* satisfying the existence of inverse.
- It is straightforward to show that if $map(G_1 \models G_2) \in Aut$ and $map(G_2 \models G_3) \in Aut$ then $map(G_1 \models G_3) \in Aut$.

Our proof method in Chapter 5 is designed to handle relative safety assertions as well, such that we do not only specify and use for reduction, but also prove the relative safety assertions, in contrast to earlier approaches to symmetry in verification.

4.5.2 More Examples of Program Symmetry

We now proceed with more examples to demonstrate the expressiveness of relative safety assertions.

Example 4.10. (Symmetry in Bridge Crossing TSA) The symmetry definition that we used for the Bridge Crossing Problem (Example 3.8 in Section 3.6.4), generalized to N train automata is as follows:

$$\begin{aligned} & p(L_0,L_1,\ldots,L_N,X_1,\ldots,X_N,Pos_1,Pos_N,Len) \models \\ & \quad p(L_0,L_{\rho(1)},\ldots,L_{\rho(N)},X_{\rho(1)},\ldots,X_{\rho(N)},Pos_{\rho(1)},\ldots,Pos_{\rho(N)}),Len). \end{aligned}$$

where L_0 is the location id of the controller, and $L_1, \ldots, L_N$ are location ids of the trains. So here the location id of the controller as well as the variable *Len* retains their value, while other variables are permuted by some permutation ρ. For instance, the symmetry definition for Bridge Crossing Problem with two trains can be represented as follows:

$$p(L_0, L_1, L_2, X_1, X_2, Pos_1, Pos_2, Len) \models$$
$$p(L_0, L_2, L_1, X_2, X_1, Pos_2, Pos_1, Len).$$

Example 4.11. (Rotational Symmetry) Here we demonstrate *rotational* symmetry in the solution of N dining philosophers' problem using $N-1$ tickets [25]. For simplicity, we assume there are $N = 3$ philosophers having ids 1, 2, and 3, and there are three forks represented as boolean variables $f_1,\ f_2,\ f_3$ which are forks used by philosophers 3 and 1, 1 and 2, and 2 and 3, respectively. Initially the ticket number $t = 2$. The program code of Philosopher 1 is shown as Program 4.17, and its CLP model as Program 4.18. A philosopher "eats" at program point $\langle 3 \rangle$, and "thinks" at program point $\langle 0 \rangle$. For our purpose it is suffice to demonstrate the rotational symmetry as the assertion:

$$p(L_1, L_2, L_3, F_1, F_2, F_3, T) \models p(L_3, L_1, L_2, L_3, L_1, L_2, T),$$

where L_i denotes the program point of philosopher i, F_i the value of f_i, and T is the number of tickets left. The above assertion specifies one cyclic shift to the right. Any cyclic shifts can be represented by the composition of this cyclic shift.

For this example, arbitrary transposition does not result in automorphism. For example, the program may be in the state $l_1 = 3, l_2 = 0, l_3 = 0, f_1 = 1, f_2 = 1, f_3 = 0, t = 1$ (that is, Philosopher 1 is eating while both philosophers 2 and 3 are thinking). Now, arbitrarily exchanging Philosopher 2 and 3, and forks 2 and 3 results in the state $l_1 = 3, l_2 = 0, l_3 = 0, f_1 = 1, f_2 = 0, f_3 = 1, t = 1$, which specifies that Philosopher 1 is eating with 1 fork, while either Philosopher 2 or 3 is thinking holding 1 fork. We know that this state is unreachable.

```
Initially $t = 2$ and $f_i = 0$ for all $i$.
      loop forever
⟨0⟩       await $(t > 0)$ $t := t - 1$
⟨1⟩       await $(f_1 = 0)$ $f_1 := 1$
⟨2⟩       await $(f_2 = 0)$ $f_2 := 1$
          // eating
⟨3⟩       $f_1 := 0$
⟨4⟩       $f_2 := 0$
⟨5⟩       $t := t + 1$
      end loop
```

Program 4.17: Philosopher 1

```
p(0,0,0,0,0,0,2).

p(1,L2,L3,F1,F2,F3,T-1)  :-  p(0,L2,L3,F1,F2,F3,T),T > 0.
p(2,L2,L3,1,F2,F3,T)  :-  p(1,L2,L3,0,F2,F3,T).
p(3,L2,L3,F1,1,F3,T)  :-  p(2,L2,L3,F1,0,F3,T).
p(4,L2,L3,0,F2,F3,T)  :-  p(3,L2,L3,F1,F2,F3,T).
p(5,L2,L3,F1,0,F3,T)  :-  p(4,L2,L3,F1,F2,F3,T).
p(0,L2,L3,F1,F2,F3,T+1)  :-  p(5,L2,L3,F1,F2,F3,T).
```

Program 4.18: Philosopher 1 CLP Model

Example 4.12. (Rotational Symmetry in Dining Philosophers with Timeout) Our real-time dining philosophers example in Section 3.6.4 also enjoyed similar rotational symmetry:

$$p(L_1,L_2,L_3,F_1,F_2,F_3,X_1,X_2,X_3) \models p(L_3,L_1,L_2,F_3,F_1,F_2,X_3,X_1,X_2).$$

Full permutation of process indices is also not applicable for this example.

Example 4.13. (Permutation of Variable-Value Pair) In Section 3.6.4 we have discussed a TSA version of Fischer's algorithm, which is a timing-based mutual exclusion algorithm. Consider now the program version of the two-process Fischer's mutual exclusion algorithm (see e.g. [1]) shown as Program 4.19, where the decision of which process should enter the critical section is made after a delay of 4 time units (other statements take 1 time unit each to execute).

The CLP model of Process 1 is shown as Program 4.20 (Process 2 is similar). This example uses timing, and so we implemented it using auxiliary variables T_1 and T_2 (see Section 3.4). Our

```
Initially $k = 0$
Process 1:                              Process 2:
        loop forever                            loop forever
$\langle 0 \rangle$    await $(k = 0)$              $\langle 0 \rangle$    await $(k = 0)$
$\langle 1 \rangle$    $k$ := 1                     $\langle 1 \rangle$    $k$ := 2
$\langle 2 \rangle$    delay (4)                    $\langle 2 \rangle$    delay(4)
$\langle 3 \rangle$    if $(k \neq 1)$ then         $\langle 3 \rangle$    if $(k \neq 2)$ then
          goto $\langle 0 \rangle$                           goto $\langle 0 \rangle$
        end if                                  end if
$\langle 4 \rangle$    $k$ := 0                     $\langle 4 \rangle$    $k$ := 0
        end loop                                end loop
```

Program 4.19: Two-Process Fischer's Algorithm

```
$p(0,0,0,0,0).$

$p(0,L_2,T_1',T_2,K)$ :- $p(0,L_2,T_1,T_2,K), T_1' \geq T_2, T_1 \leq T_2.$
$p(1,L_2,T_1+1,T_2,K)$ :- $p(0,L_2,T_1,T_2,K), T_1 \leq T_2.$
$p(2,L_2,T_1+1,T_2,1)$ :- $p(1,L_2,T_1,T_2,K), T_1 \leq T_2.$
$p(3,L_2,T_1+4,T_2,K)$ :- $p(2,L_2,T_1,T_2,K), T_1 \leq T_2.$
$p(0,L_2,T_1+1,T_2,K)$ :- $p(3,L_2,T_1,T_2,K), T_1 \leq T_2, K \neq 1.$
$p(4,L_2,T_1+1,T_2,K)$ :- $p(3,L_2,T_1,T_2,K), T_1 \leq T_2, K = 1.$
$p(0,L_2,T_1+1,T_2,0)$ :- $p(4,L_2,T_1,T_2,K), T_1 \leq T_2.$
```

Program 4.20: Two-Process Fischer's Algorithm CLP Model

assertion for symmetry here is

$$p(L_1,L_2,T_1,T_2,K) \models p(L_2,L_1,T_2,T_1,K'), \phi,$$

where the constraint ϕ constrains (K,K') to $(0,0),(1,2)$ or $(2,1)$. This is called *permutation of variable-value pair* in [182] since it maps the value of a variable onto a new value. This is not covered by some previous approaches to symmetry such as [107, 183]. The same permutation also applies to Fischer's Algorithm TSA of Section 3.6.4.

Generalizing the symmetry for N processes, we have the generic assertion

$$p(L_1,\ldots,L_N,X_1,\ldots,X_N,K) \models p(L_{\rho(1)},\ldots,L_{\rho(N)},X_{\rho(1)},\ldots,X_{\rho(N)},\rho(K)),$$

where $\rho(x)$ represents a permutation on indices $\{0,\ldots,N\}$, with the condition that $\rho(0) = 0$.

```
Initially $x_1 = x_2 = 0$.
Process 1:                                  Process 2:
      loop forever                                loop forever
⟨0⟩      await $(x_2 \neq 1)$ $x_1$ := 1      ⟨0⟩      $x_2$ := 1
⟨1⟩      await $(x_2 \neq 2)$ $x_1$ := 2      ⟨1⟩      await $(x_1 = 0)$ $x_2$ := 2
⟨2⟩      $x1$ := 0                            ⟨2⟩      $x_2$ := 0
      end loop                                    end loop
```

Program 4.21: Priority Mutual Exclusion

```
$p(0,0,0,0)$.

$p(1,L_2,1,X_2)$  :-  $p(0,L_2,X_1,X_2), X_2 \neq 1$.
$p(2,L_2,2,X_2)$  :-  $p(1,L_2,X_1,X_2), X_2 \neq 2$.
$p(0,L_2,0,X_2)$  :-  $p(2,L_2,X_1,X_2)$.

$p(L_1,1,X_1,1)$  :-  $p(L_1,0,X_1,X_2)$.
$p(L_1,2,X_1,2)$  :-  $p(L_1,1,X_1,X_2), X_1 = 0$.
$p(L_1,0,X_1,0)$  :-  $p(L_1,2,X_1,X_2)$.
```

Program 4.22: Priority Mutual Exclusion CLP Model

Example 4.14. (Priority Mutual Exclusion) We can also express the kind of "not-quite" symmetry, as exemplified by the simple two-process priority mutual exclusion of Program 4.21. The existence of priority among processes usually destroys symmetry, but not with our approach.

In Program 4.21, each process has ⟨2⟩ as the critical section. Initially, the values of both x_1 and x_2 are 0. The CLP model is Program 4.22. This example is semantically similar to the asymmetric readers-writers in [54] and the priority mutual exclusion in [182]. Although the state graph of the program is not symmetric, the state space, i.e. the set of nodes in the state graph, is, and knowing this is already useful for search space reduction in proving safety properties such as mutual exclusion. We can represent the symmetry on the reachable states simply as:

$$p(L_1,L_2,X_1,X_2) \models p(L_2,L_1,X_2,X_1). \tag{4.7}$$

It is not immediately obvious that this symmetry holds based on syntactic observation alone.

We now explain in more detail how our approach able to capture the automorphism of this "not-quite" symmetric example. Let us first examine the state graph of the program in Figure 4.2. In previous approaches to symmetry in verification, we are forced to distinguish between state (A) and (B) since (A) has an outgoing edge that reaches state (C), while state (B) does not.

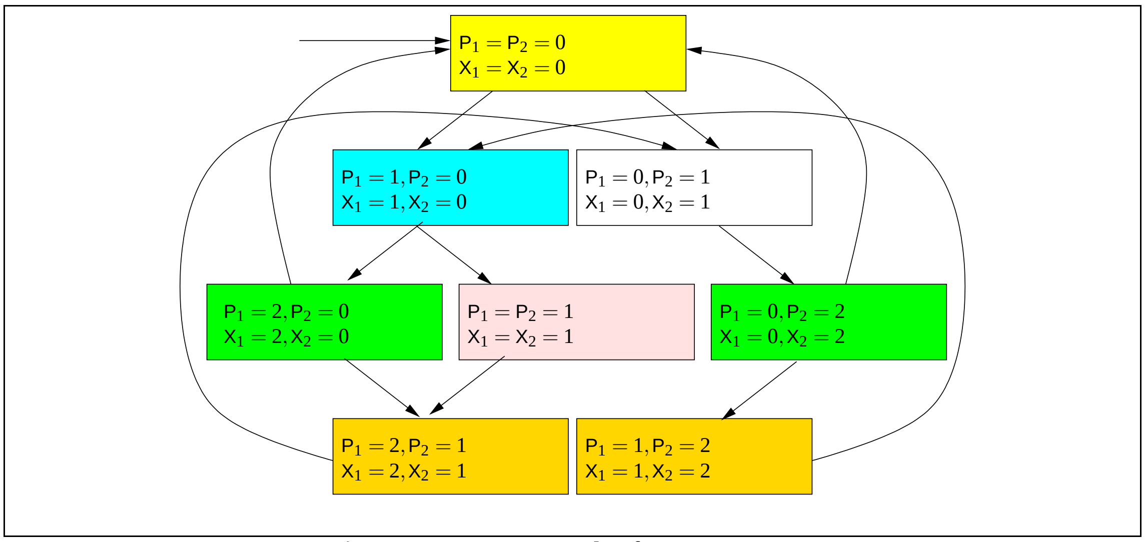

Figure 4.2: State Graph of Priority Mutex

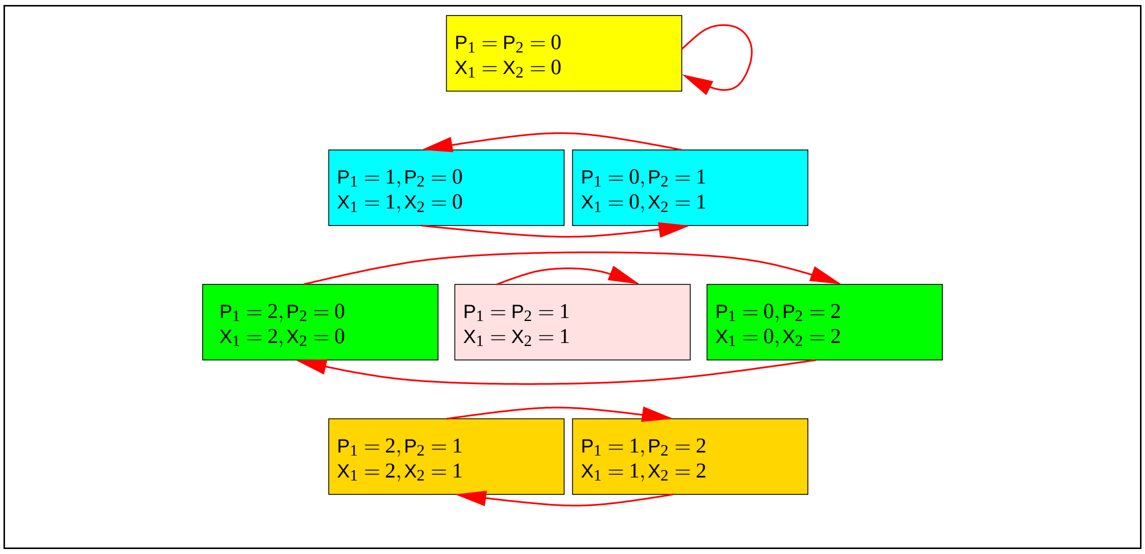

Figure 4.3: Automorphisms on Collecting Semantics

The reason for considering edges here is because of the need to verify temporal logic properties, which include liveness. Restricting ourselves to safety properties, however, allows us to blur distinctions due to edges. This is because safety properties only concern the set of reachable states (*collecting semantics*). This we clarify using Figure 4.3, where states (A) and (B) are now drawn with the same color. The arrows in Figure 4.3 shows the automorphisms in the reachable states of the program, which can exactly be represented as the relative safety assertion (4.7).

Example 4.15. (Szymanski's Algorithm) Szymanski's algorithm is a more complex priority-based mutual exclusion algorithm which is commonly encountered in the literature. We show the pseudo code as Program 4.23, where program point $\langle 8 \rangle$ is the critical section. Its CLP model is Program 4.24.

```
Initially $x_1 = x_2 = 0$.
Process 1:                                    Process 2:
        loop forever                                  loop forever
⟨0⟩         $x_1$ := 1                        ⟨0⟩         $x_2$ := 1
⟨1⟩         await ($x2 < 3$)                  ⟨1⟩         await ($x_1 < 3$)
⟨2⟩         $x_1$ := 3                        ⟨2⟩         $x_2$ := 3
⟨3⟩         if ($x_2 = 1$) then               ⟨3⟩         if ($x_1 = 1$) then
⟨4⟩             $x_1$ := 2                    ⟨4⟩             $x_2$ := 2
⟨5⟩             await ($x_2 = 4$)             ⟨5⟩             await ($x_1 = 4$)
            end if                                        end if
⟨6⟩         $x_1$ := 4                        ⟨6⟩         $x_2$ := 4
⟨7⟩         skip                              ⟨7⟩         await ($x_1 < 2$)
⟨8⟩         await ($x_2 < 2 \vee x_2 > 3$)    ⟨8⟩         skip
⟨9⟩         $x_1$ := 0                        ⟨9⟩         $x_2$ := 0
        end loop                                      end loop
```

Program 4.23: Two-Process Szymanski's Algorithm

Since the algorithm is based on prioritizing Process 1 to enter the critical section ⟨8⟩, it is not possible for Process 2 to be in the critical section while Process 1 is trying to enter the critical section. For example, the following simple symmetry does not hold:

$$p(8,7,X_1,X_2) \models p(7,8,X_2,X_1).$$

Here, some states satisfying $l_1 = 8, l_2 = 7$ are reachable, and no reachable state satisfies $l_1 = 7, l_2 - 8$. Therefore, a simple symmetry assertion such the one given in the bakery algorithm does not hold.

However, the following "not-quite" symmetry assertions still hold:

$$\begin{aligned}
p(8,L_2,X_1,X_2), L_2 < 3 &\models p(L_2,8,X_2,X_1).\\
p(8,L_2,X_1,X_2), L_2 > 7 &\models p(L_2,8,X_2,X_1).\\
p(9,L_2,X_1,X_2), L_2 \neq 7 &\models p(L_2,9,X_2,X_1).\\
p(L_1,L_2,X_1,X_2), L_1 \neq 8, L_1 \neq 9 &\models p(L_2,L_1,X_2,X_1).
\end{aligned}$$

At first it seems that the above assertions no longer define an automorphism group since $p(L_1,8,X_1,X_2), 3 \leq L_1 \leq 7 \models p(8,L_1,X_2,X_1)$ can be derived from the last assertion, yet the inverse does not hold. However, by observation the assertion $p(L_1,8,X_1,X_2) \models L_1 < 3 \vee L_1 > 7$ holds since it is not possible for Process 2 to be in the critical section while process 1 is waiting and therefore the goal $p(L_1,8,X_1,X_2) \models 3 \leq L_1 \leq 7$ does not represent any reachable state. Similarly,

```
p(0,0,0,0).

p(1,L2,1,X2) :- p(0,L2,X1,X2).
p(2,L2,X1,X2) :- p(1,L2,X1,X2),X2 < 3.
p(3,L2,3,X2) :- p(2,L2,X1,X2).
p(4,L2,X1,X2) :- p(3,L2,X1,X2),X2 = 1.
p(5,L2,2,X2) :- p(4,L2,X1,X2).
p(6,L2,X1,X2) :- p(3,L2,X1,X2),X2 ≠ 1.
p(6,L2,X1,X2) :- p(5,L2,X1,X2).
p(7,L2,4,X2) :- p(6,L2,X1,X2).
p(8,L2,X1,X2) :- p(7,L2,X1,X2).
p(9,L2,X1,X2) :- p(8,L2,X1,X2),(X2 < 2 ∨ X2 > 3).
p(0,L2,0,X2) :- p(9,L2,X1,X2).

p(L1,1,X1,1) :- p(L1,0,X1,X2).
p(L1,2,X1,X2) :- p(L1,1,X1,X2),X1 < 3.
p(L1,3,X1,3) :- p(L1,2,X1,X2).
p(L1,4,X1,X2) :- p(L1,3,X1,X2),X1 = 1.
p(L1,5,X1,2) :- p(L1,4,X1,X2).
p(L1,6,X1,X2) :- p(L1,3,X1,X2),X1 ≠ 1.
p(L1,6,X1,X2) :- p(L1,5,X1,X2).
p(L1,7,X1,4) :- p(L1,6,X1,X2).
p(L1,8,X1,X2) :- p(L1,7,X1,X2),X1 < 2.
p(L1,9,X1,X2) :- p(L1,8,X1,X2).
p(L1,0,X1,0) :- p(L1,9,X1,X2).
```

Program 4.24: Two-Process Szymanski's Algorithm CLP Model

$p(L_1,9,X_1,X_2) \models L_1 \neq 7$ also holds. These impose restrictions on the last assertion above.

We are not aware of any verification technique that would allow us to express and use this kind of symmetry.

Example 4.16. (Commutativity) Now consider the simplified two-process concurrent program shown as Program 4.25. Its CLP model contains at least the following two clauses:

$$p(6,L_2,X+3) \;\texttt{:-}\; p(5,L_2,X).$$
$$p(L_1,12,X-5) \;\texttt{:-}\; p(L_1,11,X).$$

In Program 4.25, the statement at $\langle 5\rangle$ of Process 1 is commutative to the statement at $\langle 11\rangle$ of Process 2, since from any state where Process 1 is at $\langle 5\rangle$ and Process 2 is at $\langle 6\rangle$, executing either statement first before the other results in the same state. In other words, the state $l_1 = 6, l_2 = 12, x = \mu$ for any μ is reachable only if state $l_1 = 5, l_2 = 11, x = \mu + 2$ is[2]. The commutativity

[2]The reverse also holds, but not considered here.

```
Process 1:                     Process 2:
        ⋮                              ⋮
⟨5⟩   x := x+3                 ⟨11⟩   x := x−5
⟨6⟩     ⋮                      ⟨12⟩     ⋮
```

Program 4.25: Commutative Concurrent Program

property here can be expressed by the assertion

$$p(6,12,X) \models p(5,11,X+2).$$

Example 4.17. (Serializability) Here we discuss another application of relative safety assertion beyond symmetry. We show a producer/consumer program as Program 4.26, whose CLP model is given as Program 4.27.

The macros $con_k()$ and $pro_l()$, abstract program fragments that serve to produce and consume respectively. We will imagine that apart from the variable *full* there is another variable x which may be used in $con_k()$ and $pro_l()$.

Consider the assertions:

$$p(n+1,L_2,Full,f(X)),L_2 \leq n \models p(1,L_2,Full,X).$$
$$p(L_1,n,Full,g(X)),L_1 \geq 1 \models p(L_1,0,Full,X),$$

where the expression $f(X)$ and $g(X)$ are the results of performing $con_1(),\ldots con_n()$ and $pro_1()\ldots pro_n()$ respectively on x. Then the assertions say that the result of performing the interleaving of $con_k()$ and $pro_l()$ macros, $1 \leq k \leq L_1 - 1$, $1 \leq l \leq L_2$ is as though the two sequences of transitions are serializable. Note that here, although hardly a case of symmetry, we also have an automorphism group which contains the mappings of reachable states defined by the above assertions and their inverses.

```
Initially $full$ := 0.
Consumer:                                   Producer:
          loop forever                                loop forever
    ⟨0⟩     await ($full = 1$) $full$ := 0        ⟨0⟩     $pro_1()$
    ⟨1⟩     $con_1()$                                    ⋮
              ⋮                               ⟨n−1⟩     $pro_n()$
    ⟨n⟩     $con_n()$                             ⟨n⟩     await ($full = 0$) $full$ := 1
  ⟨n+1⟩     skip                                ⟨n+1⟩     skip
          end loop                                    end loop
```

Program 4.26: Producer/Consumer

```
p(0,0,0,X).

p(1,L2,0,X) :- p(0,L2,1,X).
p(2,L2,Full,X) :- p(1,L2,Full,X).
p(n,L2,Full,X) :- p(n-1,L2,Full,X).
p(0,L2,Full,X) :- p(n,L2,Full,X).

p(L1,1,Full,X) :- p(L1,0,Full,X).
p(L1,n,Full,X) :- p(L1,n-1,Full,X).
p(L1,n+1,Full,X) :- p(L1,n,Full,X).
p(L1,0,1,X) :- p(L1,n+1,0,X).
```

Program 4.27: Producer/Consumer Partial CLP Model

4.6 Discussions

4.6.1 Liveness

So far we have discussed how to use our assertions to specify program properties that belong to the safety class. Actually, we can also use our assertion language to specify some form of liveness properties. Let us now re-visit our first example Program 3.1 in Chapter 3, with its CLP model Program 3.2. We may write an assertion such as the following:

$$S = (X^2 - X)/2, 0 \leq X, X = N \models p(\Omega, X, S, N).$$

The meaning of the assertion is that the least model of Program 3.2 necessarily includes all $p(\Omega, x, s, n)$ such that $s = (x^2 - x)/2$, $0 \leq x$, and $x = n$. When we reflect this onto the original Program 3.1, the assertion says that the state where $l = \Omega$, $s = (x^2 - x)/2$, $0 \leq x$, and $x = n$ is

```
s(ω,ω).
s(X,X_f)  :-  X ≠ ω, p(X) = 1, X_f = X.
s(X,X_f)  :-  X ≠ ω, p(X) = 0, s(h(X), ?Y), s(?Y, X_f).
```

Program 4.28: Example 12 of [135]

reachable. Adding the traditional safety assertion

$$p(\Omega,X,S,N) \models S = (X^2 - X)/2, 0 \leq X, X = N,$$

we obtain an equivalence which specifies that the end program point Ω is both possible (reachable), and also that it is necessarily reached with $s = (x^2 - x)/2$, $0 \leq x$, and $x = n$. Here we are saying that at any point in the program's execution, it is always possible for the program to reach such end state. We do not, however, say whether such end state will be reached in a finite amount of time.

4.6.2 General Equivalence

In the previous section we have discussed a use of equivalence to specify some liveness property. Equivalences can be specified as a two-way implications. Here we discuss another example from the paper of Manna et al. [135], which contains examples of properties of recursive program schema containing equivalence.

One example program schema is as follows (Example 12 in [135]):

$$F(x) \Leftarrow \textbf{ if } p(x) \textbf{ then } x \textbf{ else } F(F(h(x))) \textbf{ end if}$$

Without explaining formally how we translate recursive program schema of Manna et al. into CLP, we simply provide the CLP model of the schema as Program 4.28 according to the semantics given in [135].

Here specify the idempotence of the program schema, that is, $F(F(X)) \equiv F(X)$. Given Program 4.28, we can state the same property as:

$$s(X,Y), s(Y,X_f) \Leftrightarrow s(X,X_f).$$

We can rewrite the above formula into a conjunction of the following two assertions:

$$s(X,Y), s(Y,X_f) \models s(X,X_f)$$
$$s(X,X_f) \models s(X,Y), s(Y,X_f).$$

We will demonstrate the proof of the assertions in the next chapter.

Chapter 5

A Proof Method

In this chapter we explicate on how to prove the assertions which we have discussed extensively in Chapter 4. We first start with a motivating example. We then proceed with a few basic definitions and explain the outline of our proof method, which is based on a search process to discover premises which establish the assertion. We then detail our proof rules, and demonstrate some examples before providing their formal proofs. We then proceed on discussing extensions of our proof method for handling various verification problems. These include a intermittent abstraction mechanism, program verification technique, reduction techniques which include symmetry reduction, and the verification of recursive data structures. We end this chapter after presenting some discussions and related work on CLP-based proof methods.

5.1 First Example

Let us first examine Program 5.1, and consider how we may prove that

$$p(X) \models X = 2 \times ?Y.$$

Given a grounding σ of X, this assertion is intuitively true, since $p(X\sigma)$ is only true when $X\sigma$ is an even number.

The assertion can be deduced from the following three premises using natural deduction:

1. $\langle \forall X : p(X) \Leftrightarrow (X = 0 \vee p(X-2)) \rangle$,

2. $\langle \forall X : p(X-2) \Rightarrow \langle \exists Y : X = 2Y \rangle \rangle$,

3. $\langle \forall X : X = 0 \Rightarrow \langle \exists Y : X = 2Y \rangle \rangle$.

```
p(0).                    κ1
p(X+2)  :-  p(X).        κ2
```

Program 5.1: Even Number Generator

The first premise is the Clark completion (Section 2.7) of our example program. For the moment, the latter two we assume as given. The natural deduction proof using these premises is shown in Figure 5.1[1].

Unfortunately, the proof is not established without also demonstrating the premises. Premise 1 is given by the CLP program, hence it is given. Further, we can easily see that Premise 3 is true.

Notice that Premise 2 is in some sense "similar" to the original assertion. This could therefore lead to infinite reasoning since by applying the same proof steps to Premise 2 we would again need to establish a premise "similar" to the original assertion and Premise 2. Here we therefore need to employ a form of induction.

Recall that we want to establish the original assertion $p(X) \models X = 2\times?Y$. For this purpose, we first hypothesize that the assertion holds when we replace X with $X-2$, that is, $p(X-2) \models X-2 = 2\times?Y$. Now, assuming $p(X-2) \models X-2 = 2\times?Y$ holds, can we inductively prove that $p(X) \models X = 2\times?Y$? We certainly can, since $p(X-2) \models X-2 = 2\times?Y$ is just equivalent to $p(X-2) \models X = 2\times?Y$, which is Premise 2. That is, here we derive Premise 2 immediately from an induction hypothesis, in the context of an inductive proof of the original assertion.

In the subsequent sections we will explain a systematic proof method to obtain the premises and to apply inductive reasoning. The method proves general assertions of the form $G \models H$ for goals G and H. We begin in the next section with a few definitions.

5.2 Basic Definitions

Definiton 5.1 (Unfold). Given a CLP program Γ and a goal G which contains an atom A, a *complete unfold* of a goal G, denoted by $\mathit{unfold}_A(G)$ is the set

$$\{G' | \langle \exists \kappa \in \Gamma : \square \ \text{:-}\ G' \equiv \mathit{resolv}_A(\square \ \text{:-}\ G, \kappa)\rangle\}.$$

[1]For an introduction to natural deduction, see [120].

1	$\langle \forall X : p(X) \Leftrightarrow (X = 0 \vee p(X-2)) \rangle$	
4	$\cancel{p(i)}$	
5	$p(i) \Leftrightarrow (i = 0 \vee p(i-2))$	$\forall E1$
6	$p(i) \Rightarrow (i = 0 \vee p(i-2))$	$\wedge E5$
7	$i = 0 \vee p(i-2)$	$\Rightarrow E4,6$
2	$\langle \forall X : p(X-2) \Rightarrow \langle \exists Y : X = 2Y \rangle \rangle$	
8	$\cancel{p(i-2)}$	
9	$p(i-2) \Rightarrow \langle \exists Y : i = 2Y \rangle$	$\forall E2$
10	$\langle \exists Y : i = 2Y \rangle$	$\Rightarrow E8,9$
3	$\langle \forall X : X = 0 \Rightarrow \langle \exists Y : X = 2Y \rangle \rangle$	
11	$\cancel{i = 0}$	
12	$i = 0 \Rightarrow \langle \exists Y : i = 2Y \rangle$	$\forall E3$
13	$\langle \exists Y : i = 2Y \rangle$	$\Rightarrow E11,12$
14	$\langle \exists Y : i = 2Y \rangle$	$\vee E7,10,13$
15	$p(i) \Rightarrow \langle \exists Y : i = 2Y \rangle$	$\Rightarrow I6,15$
16	$\langle X : p(X) \Rightarrow \langle \exists Y : X = 2Y \rangle \rangle$	$\forall I15$

Figure 5.1: $p(X) \models X = 2 \times ?Y$ Natural Deduction Proof

We now state the *logical semantics* of unfold.

Proposition 5.1. $\langle \forall \tilde{X}, \tilde{Y}, \tilde{Z} : G \Leftrightarrow \bigvee \mathit{unfold}_{p(\tilde{X})\sigma}(G) \rangle$.

Proof. When A is $p(\tilde{X})\sigma$, where σ is some (not necessarily ground) substitution, the formula $\bigvee \mathit{unfold}_A(G)$ is basically the result of application of modus ponens using the Clark completion of the predicate p defined in the program Γ. When the completion of $p(\tilde{X})$ in Γ is $\langle \forall \tilde{X}, \tilde{Y} : p(\tilde{X}) \Leftrightarrow B_1 \vee \ldots \vee B_n \rangle$ for some goals B_i, $1 \leq i \leq n$, we have for any goal H with free variables from the set $\tilde{X} \cup \tilde{Z}$, $\langle \forall \tilde{X}, \tilde{Y}, \tilde{Z} : p(\tilde{X})\sigma, H \Leftrightarrow (B_1 \vee \ldots \vee B_n)\sigma, H \rangle$. If, without loss of generality, we assume that the goal G is $p(\tilde{X})\sigma, H$, then by the above definition of unfold, the formula $\bigvee \mathit{unfold}_{p(\tilde{X})\sigma}(G)$ is equivalent to $(B_1\sigma, H) \vee \ldots \vee (B_n\sigma, H)$. Hence the proposition holds. □

To understand the use of an unfold in a proof process, let us reconsider our natural deduction proof of $p(X) \models X = 2 \times ?Y$ in Section 5.1. We could obtain the Premises 2 and 3 from an unfold of $p(X)$. Given Program 5.1, $\bigvee \mathit{unfold}_{p(X)}(p(X)) \equiv X = 0 \vee p(X-2)$ and since $p(X) \Leftrightarrow X = 0 \vee p(X-2)$ (that is, Premise 1), $p(X) \models X = 2 \times ?Y$ holds if and only if $X = 0 \vee p(X-2) \models X = 2 \times ?Y$. This holds when $p(X-2) \models X = 2 \times ?Y$ (Premise 2) and $X = 0 \models X = 2 \times ?Y$ (Premise 3).

In the proof process to be introduced later, we will deal with relations between assertions. In our proof method, the proof of an assertion can be replaced with the proof of a *stronger* assertion.

We start with a formalization of what we mean by stronger or weaker assertion.

Definiton 5.2 (Assertion Entailment). An assertion $G' \models H'$ *entails* another assertion $G \models H$ written $(G' \models H') \rhd (G \models H)$ if and only if $(G' \models H') \Rightarrow (G \models H)$. In this case we say that $(G' \models H')$ is *stronger* than $(G \models H)$ and $(G \models H)$ is *weaker* than $(G' \models H')$.

Entailment is useful especially in an inductive proof, where we apply an induction whenever we discover an assertion that is entailed by an ancestor assertion. However, here we need an effective way to establish entailment.

It is easy to see that $(G' \models H') \rhd (G \models H)$ holds if and only if

$$G, \neg H \Rightarrow (G', \neg H')\sigma \tag{5.1}$$

for some substitution σ, which holds if and only if both of the following formulas hold:

1. $G, \neg H \Rightarrow G'\sigma$. To establish this formula it is sufficient to prove that $G \models G'\sigma$, which is called *subsumption*.

2. $G, \neg H \Rightarrow \neg H'\sigma$. To establish this formula it is sufficient to prove either $G, H'\sigma \models H$, $H'\sigma \models H$, or if G is of the form G'', ϕ where ϕ a constraint, $\phi, H'\sigma \models H$. We refer to any of these as *residual* obligation or assertion.

5.3 Outline of the Proof Method

In this section we will explain the outline of our proof method. We first provide a few formal definitions which will be useful for our explanation in this section.

Definiton 5.3 (Unfold Tree Goals). Given a program Γ and a set S of goals, we define the nondeterministic function $\delta(H) \subseteq H \cup \mathit{unfold}_A(G)$, when $G \in S$ and the predicate of the atom A is defined by Γ. Now, $\delta^n(\{G\})$ for some finite n is the *unfold tree goals* of G.

Proposition 5.2. Whenever $G' \in \delta^n(\{G\})$ for some n, we have that $G' \Rightarrow G$.

Proof. Whenever $\square$:- $G' \equiv resolv_{p(\tilde{X})\sigma}(\square$:- $G, \kappa)$, necessarily $G' \Rightarrow G$ since suppose that G is $p(\tilde{X})\sigma, H$ where H is a goal with free variables in the set $\tilde{X} \cup \tilde{Y}$, and κ is

$$p(\tilde{X}) \text{ :- } L_1, \ldots, L_n,$$

Then $\langle \forall \tilde{X}, \tilde{Y} : p(\tilde{X})\sigma, H \Leftarrow (L_1, \ldots, L_n)\sigma, H \rangle$. Hence, $\langle \forall \tilde{X}, \tilde{Y} : G' \Rightarrow G \rangle$. $\square$

Definiton 5.4 (Unfold Frontier). Given a CLP program Γ and a set S of goals we define the nondeterministic function

$$\varepsilon(S) = (S - \{G'\}) \cup unfold_A(G') \text{ for some } G' \in S \text{ and } A \in G'$$

We define an *unfold frontier* of G as $\varepsilon^n(\{G\})$ for some $n \geq 1$.

Proposition 5.3. $G \Rightarrow \bigvee \varepsilon^n(G)$ for any n.

Proof. Induction using the definition of ε, and using the logical semantics of unfold. $\square$

In order to prove $G \models H$, we proceed as follows: unfold G completely a finite number of steps in order to obtain an unfold frontier containing the goals $G_1, \ldots, G_n \in \varepsilon^k(G)$ for some k. Then unfold H, but this time not necessarily completely, that is, we simply obtain some unfold tree goals of H, which are $H_1, \ldots, H_m \in \delta^l(H)$ for some l. This situation is depicted in Figure 5.2. Then, the proof holds if

$$G_1 \vee \ldots \vee G_n \models H_1 \vee \ldots \vee H_m$$

or alternatively, $G_i \models H_1 \vee \ldots \vee H_m$ for all $1 \leq i \leq n$. This follows easily from the fact that $G \models G_1 \vee \ldots \vee G_n$ (since from Proposition 5.3, $G \Rightarrow \bigvee \varepsilon^k(G)$ for any k), and $H_j \models H$ for all j such that $1 \leq j \leq m$ (since from Proposition 5.2, whenever $G' \in \delta^l(\{G\})$ for some l, we have that $G' \Rightarrow G$).

More specifically, but with some loss of generality, the proof holds if

$$\langle \forall i : 1 \leq i \leq n \Rightarrow \langle \exists j : 1 \leq j \leq m \wedge G_i \models H_j \rangle \rangle.$$

and for this reason, our *proof obligation* will be constructed from an assertion such as $G_i \models H_j$.

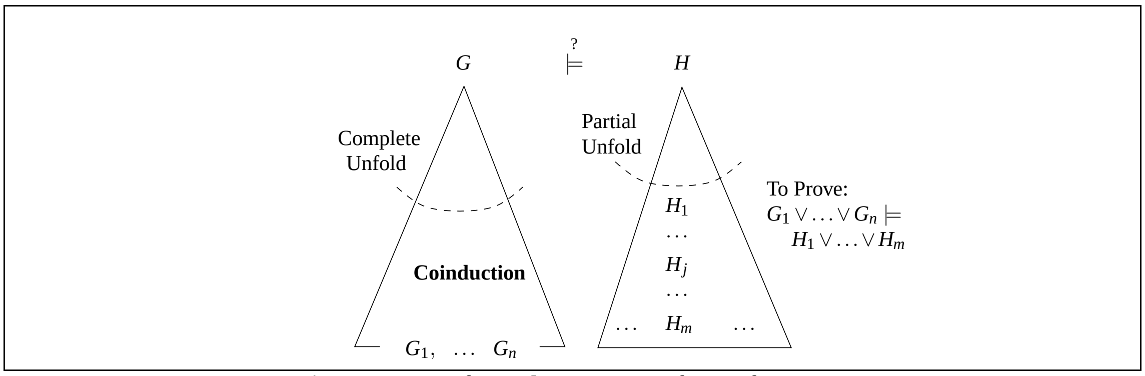
$G \overset{?}{\models} H$
Complete Unfold
Coinduction
$G_1, \ \ldots \ G_n$
Partial Unfold
H_1
H_j
H_m
To Prove:
$G_1 \vee \ldots \vee G_n \models H_1 \vee \ldots \vee H_m$

Figure 5.2: Informal Structure of Proof Process

Our proof method produces proof obligations and attempts to discharge them by direct proof using constraint solver or by inductive reasoning.

Induction is used to handle the possibly infinite unfoldings of G and H in Figure 5.2. We call our version of induction as *coinduction*. We apply an induction hypothesis whenever we discover an obligation that is entailed by some ancestor obligation. We allow all frontier assertions to be proved inductively, and this is why we use the term coinduction due to the sense of distinction between our use of induction and normal well-founded induction which requires base case (non-inductive proof).

5.4 Proof Rules

We now present a calculus for proving assertions $G \models H$.

The proof process starts with a set of proof obligations and attempts to discharge them one by one (although at times the set may in fact become larger).

Definiton 5.5 (Proof Obligation). A *proof obligation* is of the form $\tilde{A} \vdash G \models H$, where G and H are goals and $\tilde{A}$ is a set of assertions.

The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$, called the *set of assumed assertions*, contains assertions that can be used as induction hypothesis to discharge the proof obligation at hand. We sometimes also call as proof obligation the assertion part $G \models H$ of a proof obligation.

Our proof rules are presented in Figure 5.3. The $\uplus$ symbol represents the disjoint union of two sets, and emphasizes the fact that in an expression of the form $\tilde{A} \uplus \{a\}$, we have that $a \notin \tilde{A}$. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting a proof obligation from Π and attempting to discharge it. In this process, new proof obligations may be produced. The proof process is typically centered around unfolding the goals in proof obligations.

The *direct proof* (DP) rule is the simplest proof rule. It discharges a proof obligation when it can be directly proved that it holds, possibly by substituting some existentially-quantified variables of the rhs of the assertion.

The *left unfold* (LU) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as possible induction hypothesis to every newly produced proof obligation, opening the door to using induction in the proof.

The *right unfold* (RU) rule performs an unfold operation on the rhs of a proof obligation. Note that there are a number of choices for $H' \in \mathit{unfold}(H)$, and it is generally not known which H' is the one we need. As a rule of thumb, however, H' should be chosen in such a way that the new expression $G \models H'$ looks "similar" to an element of $\tilde{A}$, thus making possible the application of the AP rule later, or such that H' is "similar" to G, thus making possible the application of the DP rule.

The *assumption proof* (AP) rule transforms an obligation by using a hypothesis. Since an induction hypothesis can only be created using the LU rule from a parent goal, the AP rule realizes the coinduction principle, where we use the parent goal itself as hypothesis to prove the obligation at hand. The underlying principle behind the AP rule is that a similar assertion $G' \models H'$ has been previously encountered in the proof process, and is to be proved. Now, that assertion can be used as an induction hypothesis in order to establish the current assertion $G \models H$ provided $(G' \models H') \rhd (G \models H)$.

The cut (CUT) rule is used for strengthening a proof obligation. I is useful mainly for generalizing the lhs goal of an assertion. Given an assertion $G \models H$, it is often the case that H is too weak to result in applications of other rules that would lead to a successful proof. To address this, the CUT rule introduces a new, goal G' possibly stronger than H, with the condition that is weaker than G (that is, $G \models G'$). The obligation to prove $G \models H$ is now replaced with the obligation to prove that G' is indeed stronger than H ($G' \models H$). This technique is employed in the following cases:

1. Abstracting a program condition into a more general description.

2. Introduction of *loop invariant*. In the program verification framework [100], a human user

$$
\begin{array}{lll}
(\textsc{dp}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G, \phi \models H\}}{\Pi} & \text{There exists a substitution } \sigma \text{ of existential variables in } H \text{ s.t. } G \models H\sigma \\[2ex]
(\textsc{lu}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G \models H\}}{\Pi \,\cup\, \bigcup_{i=1}^{n} \{\tilde{A} \cup \{G \models H\} \vdash G_i \models H\}} & \mathit{unfold}(G) = \{G_1, \ldots, G_n\} \\[2ex]
(\textsc{ru}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G \models H\}}{\Pi \cup \{\tilde{A} \vdash G \models H'\}} & H' \in \mathit{unfold}(H) \\[2ex]
(\textsc{ap}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G \models H\}}{\Pi} & G' \models H' \in \tilde{A}, (G' \models H') \rhd (G \models H) \\[2ex]
(\textsc{cut}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G \models H\}}{\Pi \cup \{\tilde{A} \vdash G' \models H'\}} & (G' \models H') \rhd (G \models H) \\[2ex]
(\textsc{spl}) & \dfrac{\Pi \uplus \{\tilde{A} \vdash G \models H\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash G, \phi_i \models H\}} & \phi_1 \vee \ldots \vee \phi_k \Leftrightarrow \neg\Box.
\end{array}
$$

Figure 5.3: Proof Rules

provides the loop invariant which generalizes the program states at a particular point within a loop. This is exemplified later in the proof of the Sum program we give in Section 5.5.

3. Translation into equivalent condition. This is useful for the reduction of proof size by the use of relative safety (e.g., symmetry), as to be explained in Section 5.6.

4. Strengthening of *inductive invariant* in the context of verification using Manna-Pnueli's universal invariance rule to be discussed in Section 5.10.2.

Finally, the *split* (SPL) rule converts a proof obligation into several, more specialized ones.

Given an assertion $G \models H$, a proof shall start with $\Pi = \{\emptyset \vdash G \models H\}$, and proceed by repeatedly applying the rules in Figure 5.3 to it. The proof finishes when there are no more obligations left to be proved. We will state the soundness of our proof rules and its proof in Section 5.7.

5.5 Proof Scope Notation and Simple Examples

We may apply the LU rule to the example of Section 5.1. Initially we have $\Pi = \{\emptyset \vdash p(X) \models X = 2\times ?Y\}$. Applying the LU rule using both CLP clauses κ_1 and κ_2, results in a set Π' of proof obligations containing the two proof obligations:

- $\{p(X) \models X = 2\times ?Y\} \vdash X = 0 \models X = 2\times ?Y$ and
- $\{p(X) \models X = 2\times ?Y\} \vdash p(X-2) \models X = 2\times ?Y$.

The first obligation can be discharged using the DP rule since $X = 0 \models X = 2\times?Y$ obviously holds. We prove the second assertion using AP. We notice that the assertion $p(X) \models X = 2\times?Y$ is in the set of assumed assertions. Here we want to prove both the subsumption and the residual obligation to establish $(p(X) \models X = 2\times?Y) \rhd (p(X-2) \models X = 2\times?Y)$. Now consider the substitution $\sigma = \{X \mapsto X-2, Y \mapsto Z\}$. Obviously, the subsumption $p(X-2) \models p(X)\sigma$ holds and can be proved by DP. Moreover, the residual obligation $X - 2 = 2 \times Z \models X = 2\times?Y$ also holds by DP (consider substituting Y with $Z+1$).

Explaining the proof in English as above is rather tedious. Therefore here we introduce our flavor of proof scope notation, in order to compactly write the proofs. We show the features of our scope notation by using it to represent the proof just explained, in Figure 5.4. Notice the followings:

- We write the original assertion to be proved above a horizontal line.
- We never write down the set of assumed assertions in the scope notation, but we assume that it is accordingly updated at every rule application.
- At the right end, we write down the explanation how the assertion has been derived. The information denotes which rule is applied to which ancestor assertions.
- The application of LU rule results in a set of assertions indexed using alphabet (e.g., the assertions 2a and 2b in Figure 5.4), each of which must eventually be discharged.
- The application of AP or CUT spawns two new obligations: the subsumption (4s.1) and residual obligation (4r.1). The numbering 4s.1 denotes the first assertion (1) in the proof of subsumption (s) to prove assertion no. 4 in the main proof. In 4r.1, r denotes residual obligation.
- We represent the discharging of an assertion by DP and AP as the formula $\neg\Box$, denoting true. Note that by rule DP and AP, a discharged assertion is removed from the set of obligations. When substitution of existential variables is necessary in the proof using DP, we provide the mapping in the explanation part.

We provide here as another example the proof of the symmetry property of the two-process bakery algorithm (Example 4.9 on Page 95). We have presented the relative safety expression that represent the symmetry as the formula (4.6) on Page 95. The complete proof is shown in both Figures 5.5 and 5.6.

1	$p(X) \models X = 2\times?Y$	
2a	$X = 0 \models X = 2\times?Y$	LU 1
2b	$p(X-2) \models X = 2\times?Y$	LU 1
3	$\neg\Box$	DP 2a
4	$\neg\Box$	AP 1,2b

4s.1	$p(X-2) \models p(X-2)$	AP 1,2b
4s.2	$\neg\Box$	DP 4s.1

4r.1	$X-2 = 2\times Z \models X = 2\times?Y$	AP 1,2b
4r.2	$\neg\Box$	DP 4r.1 $\{Y \mapsto Z+1\}$

Figure 5.4: Scope Notation Proof of First Example

For this proof, initially, $\Pi = \{\emptyset \vdash p(L_1,L_2,X,Y) \models p(L_2,L_1,Y,X)\}$, represented as the assertion 1 in Figure 5.5. Using the LU proof rule, and all the clauses κ_1 to κ_7 of Program 3.15, we perform an unfold of obligation 1 obtaining a new set of proof obligations Π', which includes proof obligations containing the assertions 2a to 2g respectively, in Figure 5.5. Let us focus on the assertion 2b. Using proof rule RU and CLP clause κ_5 of Program 3.15, we obtain assertion 5[2]. We then apply the AP rule to assertion 5 using assertion 1 as the induction hypothesis. We show the subsumption and residual obligation proofs of this AP application in Figure 5.6. Other assertions from 2c to 2g are proved similarly. In proving 2a, no hypothesis application (AP) is necessary.

As our next example, we provide the proof of the assertion

$$p(0,0,0,N,S_f), N \geq 0 \models S_f = (N^2 - N)/2$$

on Program 3.4, which, note again, is a forward CLP model of Program 3.1 with final variables. The proof is shown in Figure 5.7.

It is not possible to establish obligation 1 without using the CUT rule. CUT strengthens an assertion by weakening (generalizing) its lhs. For this purpose, we replace obligation 1 with the stronger obligation 2. In this case, we often call the lhs of obligation 2 as a *loop invariant,* because it essentially represent the loop invariant of the while loop, as to be explain in Section 5.8.3. It is called invariant because when we can complete the proof coinductively by an application of AP

[2]After performing right unfold, we may obtain existentially-quantified variables (prefixed with "?"). This is not the case with left unfold since the lhs part of an assertion is the negated part of the formula (recall that $\alpha \Rightarrow \beta$ is equivalent to $\neg\alpha \vee \beta$) such that existential quantification in the lhs is transformed to universal quantification for the whole assertion.

1	$p(L_1,L_2,X,Y) \models p(L_2,L_1,Y,X)$	
2a	$L_1 = 0, L_2 = 0, X = 0, Y = 0 \models p(L_2,L_1,Y,X)$	LU 1
2b	$L_1 = 1, X = Y+1, L_1' = 0, p(L_1',L_2,X',Y) \models p(L_2,L_1,Y,X)$	LU 1
2c	$L_1 = 2, (Y = 0 \vee X < Y), L_1' = 1, p(L_1',L_2,X,Y) \models p(L_2,L_1,Y,X)$	LU 1
2d	$L_1 = 0, X = 0, L_1' = 2, p(L_1',L_2,X',Y) \models p(L_2,L_1,Y,X)$	LU 1
2e	$L_2 = 1, Y = X+1, L_2' = 0, p(L_1,L_2',X,Y') \models p(L_2,L_1,Y,X)$	LU 1
2f	$L_2 = 2, (X = 0 \vee Y < X), L_1' = 1, p(L_1,L_1',X,Y) \models p(L_2,L_1,Y,X)$	LU 1
2g	$L_2 = 0, Y = 0, L_2' = 2, p(L_1,L_2',X,Y') \models p(L_2,L_1,Y,X)$	LU 1
3	$L_1 = 0, L_2 = 0, X = 0, Y = 0 \models L_2 = 0, L_1 = 0, Y = 0, X = 0$	RU 2a
4	$\neg\Box$	DP 3
5	$L_1 = 1, X = Y+1, L_1' = 0, p(L_1',L_2,X',Y)$ $\models L_1 = 1, X = Y+1, ?L_1'' = 0, p(L_2, ?L_1'', Y, ?X'')$	RU 2b
6	$\neg\Box$	AP 1,5
7	$L_1 = 2, (Y = 0 \vee X < Y), L_1' = 1, p(L_1',L_2,X,Y)$ $\models L_1 = 2, (Y = 0 \vee X < Y), ?L_1'' = 1, p(L_2, ?L_1'', Y, X)$	RU 2c
8	$\neg\Box$	AP 1,7
9	$L_1 = 0, X = 0, L_1' = 2, p(L_1',L_2,X',Y)$ $\models L_1 = 0, X = 0, ?L_1'' = 2, p(L_2, ?L_1'', Y, ?X'')$	RU 2d
10	$\neg\Box$	AP 1,9
11	$L_2 = 1, Y = X+1, L_2' = 0, p(L_1,L_2',X,Y')$ $\models L_2 = 1, Y = X+1, ?L_2'' = 0, p(?L_2'', L_1, ?Y'', X)$	RU 2e
12	$\neg\Box$	AP 1,11
13	$L_2 = 2, (X = 0 \vee Y < X), L_2' = 1, p(L_1,L_2',X,Y)$ $\models L_2 = 2, (X = 0 \vee Y < X), ?L_2'' = 1, p(?L_2'', L_1, Y, X)$	RU 2f
14	$\neg\Box$	AP 1,13
15	$L_2 = 0, Y = 0, L_2' = 2, p(L_1,L_2',X,Y')$ $\models L_2 = 0, Y = 0, ?L_2'' = 2, p(?L_2'', L_1, ?Y'', X)$	RU 2g
16	$\neg\Box$	AP 1,15

Figure 5.5: Symmetry Proof of Two-Process Bakery Algorithm

in just one more re-visit of $\langle 0 \rangle$, any possible state of the program at $\langle 0 \rangle$ necessarily satisfies the lhs of obligation 2, projected to the main variables of the predicate p. For this use of CUT to be valid, we also have to establish the subsumption 2s.1 and the residual obligation 2r.1. We also provide their proofs in Figure 5.7). Both 2s.1 and 2r.1 can be immediately discharged using DP.

To prove obligation 2, we apply LU producing 3a and 3b. 3a corresponds to the path that exits the program loop and reaches the end of the program, while 3b corresponds to the path that runs through the loop body. Here we can further apply LU to 3a producing 4, which can be immediately established by DP. In proving 3b, we apply LU twice resulting in 7.

Notice that 7 actually has the obligation 2 in its set of assumed assertions. This we can use as induction hypothesis to establish 7. The use of 2 as a hypothesis to prove 7 is justified by the proofs of the subsumption 8s.1 and the residual obligation 8r.1, where obligation 2 entails obligation 7.

$$\begin{array}{l|ll}
\text{6s.1} & L_1 = 1, X = Y+1, L_1' = 0, p(L_1', L_2, X', Y) \models p(L_1', L_2, X', Y) & \textsc{ap}\ 1{,}5 \\
\hline
\text{6s.2} & \neg\Box & \textsc{dp}\ \text{6s.1}
\end{array}$$

$$\begin{array}{l|ll}
\text{6r.1} & L_1 = 1, X = Y+1, L_1' = 0, p(L_2, L_1', Y, X') & \\
 & \quad \models L_1 = 1, X = Y+1, ?L_1'' = 0, p(L_2, ?L_1'', Y, ?X'') & \textsc{ap}\ 1{,}5 \\
\hline
\text{6r.2} & \neg\Box & \textsc{dp}\ \text{6r.1}\ \{X'' \mapsto X', L_1'' \mapsto L_1'\}
\end{array}$$

Figure 5.6: Subsumption and Residual Obligation Proofs of the Symmetry Proof of the Two-Process Bakery Algorithm

We note in examples such as Sum the rhs of the obligation $S_f = (N^2 - N)/2$ is in a special form that only refers to variables that are not updated throughout the unfolding. In this case the proof of residual obligations such as 8r.1 is typically easy.

5.6 Redundancy, Global Tabling, and Symmetry Reduction

5.6.1 Redundancy and Global Tabling

There is an important principle which gives rise to optimization in the proof process, that of a *redundancy* between obligations. The essential idea is based on the observation that when $G' \models H'$ has been established in one part of the proof tree, we may use it to conclude $G \models H$ when $(G' \models H') \rhd (G \models H)$ holds.

Redundancy gives rise to *global tabling* where in the application of AP, we use as assumed assertion, not only a left-unfolding ancestor of the current obligation, but also any assertion that has been encountered by the left unfolding process.

To accommodate global tabling, we update our proof rules of Figure 5.3 into those shown in Figure 5.8. The rules in Figure 5.8 manipulates a global table $\mathcal{T}$ instead of a set of assumed assertions $\tilde{A}$ as in Figure 5.3. An important point here is that the global table $\mathcal{T}$ is independent from any obligation while the set of assumed assertions $\tilde{A}$ is a component of an obligation. As such, in Figure 5.8, the obligations no longer has $\tilde{A}$ attached and they accordingly take the same form as assertions.

We note that the rules of Figure 5.8 is more powerful than the rules of Figure 5.3 since the set of assumed assertions $\tilde{A}$ in Figure 5.3 is always a subset of the global table $\mathcal{T}$. Hence, if we manage to prove an obligation by an application of AP according to Figure 5.3, the same proof can always be done with the AP of Figure 5.8. The reverse, however, does not hold. Nevertheless, here we do not make any claim that the rules of Figure 5.8 are strictly more powerful than those

1	$p(0,0,0,N,S_f), N \geq 0 \models S_f = (N^2-N)/2$	
2	$p(0,X,S,N,S_f), S=(X^2-X)/2, X \leq N, N \geq 0$ $\models S_f = (N^2-N)/2$	CUT 1
3a	$p(\Omega,X,S,N,S_f), S=(X^2-X)/2, X=N, N \geq 0$ $\models S_f = (N^2-N)/2$	LU 2
3b	$p(1,X,S,N,S_f), X<N, S=(X^2-X)/2, X \leq N, N \geq 0$ $\models S_f = (N^2-N)/2$	LU 2
4	$S=S_f, S=(X^2-X)/2, X=N, N \geq 0 \models S_f = (N^2-N)/2$	LU 3a
5	$\neg\Box$	DP 4
6	$p(2,X,S',N,S_f), S'=S+X, X<N, S=(X^2-X)/2,$ $X \leq N, N \geq 0 \models S_f = (N^2-N)/2$	LU 3b
7	$p(0,X',S',N,S_f), X'=X+1, S'=S+X, X<N, S=(X^2-X)/2,$ $X \leq N, N \geq 0 \models S_f = (N^2-N)/2$	LU 6
8	$\neg\Box$	AP 2,7

2s.1	$p(0,X,S,N,S_f), X=0, S=0, N \geq 0$ $\models p(0,X,S,N,S_f), S=(X^2-X)/2, X \leq N, N \geq 0$	CUT 1
2s.2	$\neg\Box$	DP 2s.1

2r.1	$S_f = (N^2-N)/2 \models S_f = (N^2-N)/2$	CUT 1
2r.2	$\neg\Box$	DP 2r.1

8s.1	$p(0,X',S',N,S_f), X'=X+1, S'=S+X, X<N, S=(X^2-X)/2,$ $X \leq N, N \geq 0 \models p(0,X',S',N,S_f), S'=(X'^2-X')/2, X' \leq N, N \geq 0$	AP 2,7
8s.2	$\neg\Box$	DP 8s.1

8r.1	$S_f=(N^2-N)/2, X'=X+1, S'=S+X, X<N, S=(X^2-X)/2,$ $X \leq N, N \geq 0 \models S_f = (N^2-N)/2$	AP 2,7
8r.2	$\neg\Box$	DP 8r.1

Figure 5.7: Proof of Sum

of Figure 5.3.

5.6.2 Proof Using Redundancy

In Figure 5.9, we show a proof of the mutual exclusion property of the two-process bakery algorithm (Section 4.2). In Figure 5.9 we do not provide the proofs of the subsumptions and residual obligations. Since these are simple, we provide only some of their proofs in Figure 5.10.

In Figure 5.10, we assume that the variables X and Y take nonzero integer values. The proof uses the principle of redundancy between assertions, where we denote any redundant assertion using "RED m,n" where *m* denotes the assertion to which assertion *n* is redundant. Both RED m,n and AP m,n denote the application of AP in Figure 5.8, but differentiate them based on whether the assumed assertion *m* being used is an ancestor (AP m,n) or not (RED m,n).

$$(\textsc{dp}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \vdash \Pi} \qquad \text{There exists a substitution } \sigma \text{ of existential variables in } H \text{ s.t. } G \models H\sigma$$

$$(\textsc{lu}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \cup \{G \models H\} \vdash \Pi \,\cup\, \bigcup_{i=1}^{n}\{G_i \models H\}} \qquad \mathit{unfold}(G) = \{G_1, \ldots, G_n\}$$

$$(\textsc{ru}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \vdash \Pi \cup \{G \models H'\}} \qquad H' \in \mathit{unfold}(H)$$

$$(\textsc{ap}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \vdash \Pi} \qquad G' \models H' \in \mathcal{T}, (G' \models H') \rhd (G \models H)$$

$$(\textsc{cut}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \vdash \Pi \cup \{G' \models H'\}} \qquad (G' \models H') \rhd (G \models H)$$

$$(\textsc{spl}) \qquad \frac{\mathcal{T} \vdash \Pi \uplus \{G \models H\}}{\mathcal{T} \vdash \Pi \,\cup\, \bigcup_{i=1}^{k}\{G, \phi_i \models H\}} \qquad \phi_1 \vee \ldots \vee \phi_k \Leftrightarrow \neg\Box.$$

Figure 5.8: Proof Rules with Global Table

5.6.3 Proof Using Symmetry Reduction

In this section we present symmetry reduction using our proof method. We repeat our discussion in Chapter 1 that we advocate a methodology where we first establish the relative safety assertion we wish to prove. We then use the assertion for reduction. More specifically, we use the assertion to establish redundancy (Section 5.6.1).

As has been discussed in Chapter 1 and Section 4.5, this method is more powerful that it can handle more cases of symmetry than other approaches. This is because we only consider the verification of safety properties, allowing us extra power in handling arbitrary automorphisms on the collecting semantics.

In the Section 5.5 we have demonstrated the proof of symmetry property in the two-process bakery algorithm using our proof method. Here we use the symmetry assertion to obtain an even smaller mutual exclusion proof then that of Figure 5.9 of the two-process bakery algorithm. The reduced proof is shown in Figure 5.11.

Figure 5.11 is similar to Figure 5.9 up to assertion 11b, at which point the assertions that are not yet proved are 11b and 2b.

We apply the CUT rule to 11b obtaining 13. The subsumption and residual obligation proof for the application of CUT here is shown in Figure 5.11 as the proofs of 13s.1 and 13r.1, respectively. In the proof of 13s.1 we again use CUT rule to strengthen 13s.1 to 13s.2. Here 13s.2 is the symmetry assertion itself of the two-process bakery algorithm (see Example 4.9 on Page 95 and

1	$p(2,2,X,Y) \models \Box$	
2a	$p(1,2,X,Y),(Y=0 \vee X<Y) \models \Box$	LU 1
2b	$p(2,1,X,Y),(X=0 \vee Y<X) \models \Box$	LU 1
3a	$p(0,2,X',Y),Y=0 \models \Box$	LU 2a
3b	$p(1,1,X,Y),(Y=0 \vee X<Y),(X=0 \vee Y<X) \models \Box$	LU 2a
4a	$p(2,2,X'',Y),Y=0 \models \Box$	LU 3a
4b	$p(0,1,X',Y),(X'=0 \vee Y<X'),Y=0 \models \Box$	LU 3a
5	$\neg\Box$	AP 1,4a
6	$p(2,1,X'',Y),Y=0 \models \Box$	LU 4b
7	$p(1,1,X'',Y),Y=0 \models \Box$	LU 6
8	$p(0,1,X''',Y),Y=0 \models \Box$	LU 7
9	$p(2,1,X^{iv},Y),Y=0 \models \Box$	LU 8
10	$\neg\Box$	AP 6,9
11a	$p(0,1,X',Y),Y=0 \models \Box$	LU 3b
11b	$p(1,0,X,Y'),X=0 \models \Box$	LU 3b
12	$\neg\Box$	RED 8,11a
13	$p(1,2,X,Y''),X=0 \models \Box$	LU 11b
14	$p(1,1,X,Y''),X=0 \models \Box$	LU 13
15	$p(1,0,X,Y'''),X=0 \models \Box$	LU 14
16	$\neg\Box$	AP 11b,15
17a	$p(1,1,X,Y),(Y=0 \vee X<Y),(X=0 \vee Y<X) \models \Box$	LU 2b
17b	$p(2,0,X,Y'),X=0 \models \Box$	LU 2b
18	$\neg\Box$	RED 3b,17a
19a	$p(1,0,X,Y'),(Y'=0 \vee X<Y'),X=0 \models \Box$	LU 18b
19b	$p(2,2,X,Y''),X=0 \models \Box$	LU 18b
20	$\neg\Box$	RED 11b,19a
21	$\neg\Box$	AP 1,19b

Figure 5.9: Mutual Exclusion Proof of Two-Process Bakery Algorithm

Section 5.5). We have proved 13s.2 in Section 5.5, and it need not be proved again here. 13 is now redundant to 8 and the proof need not proceed further. We similarly prove 2b by applying symmetry assertion obtaining assertion 15, and then establish its redundancy to 8.

Notice that the proof in Figure 5.11 is much smaller than the proof in Figure 5.9. The example demonstrates that our proof method is capable of handling symmetry reduction in the verification of safety properties. Recall in Chapter 4 that our assertion language is powerful enough to specify even "not-quite" symmetry properties in concurrent programs. Our proof method can also prove such assertions, which can in turn be used for reducing the size of other proofs. We can even handle examples that are not handled by previous approaches, such as the symmetry reduction for Szymanski's mutual exclusion algorithm.

Other than symmetry reduction, our proof method can also be employed for proving and using more general relative safety assertions such as commutativity and serializability, whose

$$\begin{array}{l|ll} \text{5s.1} & p(2,2,X'',Y),Y=0 \models p(2,2,X'',Y) & \text{AP } 1,4\text{a} \\ \hline \text{5s.2} & \neg\Box & \text{DP 5s.1} \end{array}$$

$$\begin{array}{l|ll} \text{5r.1} & \Box \models \Box & \text{AP } 1,4\text{a} \\ \hline \text{5r.2} & \neg\Box & \text{DP 5r.1} \end{array}$$

$$\begin{array}{l|ll} \text{12s.1} & p(0,1,X',Y),Y=0 \models p(0,1,X',Y),Y=0 & \text{RED } 8,11\text{a} \\ \hline \text{12s.2} & \neg\Box & \text{DP 12s.1} \end{array}$$

$$\begin{array}{l|ll} \text{12r.1} & \Box \models \Box & \text{RED } 8,11\text{a} \\ \hline \text{12r.2} & \neg\Box & \text{DP 12r.1} \end{array}$$

Figure 5.10: Subsumption and Residual Obligation Proofs of the Mutual Exclusion Proof of the Two-Process Bakery Algorithm

instances we give in Example 4.16 (Page 105) and Example 4.17 (Page 106).

5.7 Correctness

5.7.1 Soundness

The condition in which a proof can be completed using the rules of Figure 5.8 (and hence, as has been argued in Section 5.6, also Figure 5.3) is stated in the following theorem.

Theorem 5.1 (Proof of Assertions). $G \models H$ holds if, starting with the proof obligation $\Pi = \{\emptyset \vdash G \models H\}$, there exists a sequence of applications of proof rules that results in $\Pi = \emptyset$.

Proof. First we start by reasoning about the *soundness* of each proof rule, that is, that the proof of the obligations in the conclusion would establish the obligation of the premise.

The rule RU is sound because by the logical semantics of unfold (Proposition 5.1 on Page 112), when $H' \in \mathit{unfold}(H)$ then $H' \models H$. Therefore, the proof of the obligation $G \models H$ can be replaced by the proof of the obligation $G \models H'$ since $G \models H'$ is stronger than $G \models H$, that is, $(G \models H') \rhd (G \models H)$.

Similarly, the rule DP is sound because $G \models H$ actually holds (assumed to be proved separately), and hence can be removed from consideration.

The rule CUT is sound because we replace an obligation $G \models H$ by a stronger obligation $G' \models H'$.

The rule SPL is sound because the proof of all of $G, \phi_i \models H$ for all i such that $1 \leq i \leq k$ is the

1	$p(2,2,X,Y) \models \Box$	
2a	$p(1,2,X,Y),(Y=0 \vee X<Y) \models \Box$	LU 1
2b	$p(2,1,X,Y),(X=0 \vee Y<X) \models \Box$	LU 1
3a	$p(0,2,X',Y),Y=0 \models \Box$	LU 2a
3b	$p(1,1,X,Y),(Y=0 \vee X<Y),(X=0 \vee Y<X) \models \Box$	LU 2a
4a	$p(2,2,X'',Y),Y=0 \models \Box$	LU 3a
4b	$p(0,1,X',Y),(X'=0 \vee Y<X'),Y=0 \models \Box$	LU 3a
5	$\neg\Box$	AP 1,4a
6	$p(2,1,X'',Y),Y=0 \models \Box$	LU 4b
7	$p(1,1,X'',Y),Y=0 \models \Box$	LU 6
8	$p(0,1,X''',Y),Y=0 \models \Box$	LU 7
9	$p(2,1,X^{iv},Y),Y=0 \models \Box$	LU 8
10	$\neg\Box$	AP 6,9
11a	$p(0,1,X',Y),Y=0 \models \Box$	LU 3b
11b	$p(1,0,X,Y'),X=0 \models \Box$	LU 3b
12	$\neg\Box$	RED 8,11a
13	$p(0,1,Y',X),X=0 \models \Box$	CUT 11b
14	$\neg\Box$	RED 8,13
15	$p(1,2,Y,X),(X=0 \vee Y<X) \models \Box$	CUT 2b
16	$\neg\Box$	RED 2a,15

13s.1	$p(1,0,X,Y'),X=0 \models p(0,1,Y',X),X=0$	CUT 11b
13s.2	$p(L_1,L_2,X,Y') \models p(L_2,L_1,Y',X)$	CUT 13s.1 (Proved)

13r.1	$\Box \models \Box$	CUT 11b
13r.2	$\neg\Box$	DP 13r.1

Figure 5.11: Reduced Mutual Exclusion Proof of Two-Process Bakery Algorithm

proof of $G,(\phi_1 \vee \vdots \vee \phi_k) \models H$ which is equivalent to $G \models H$ by the side condition of the rule.

The rule LU is *partially sound* in the sense that when $unfold(G) = \{G_1,\ldots,G_n\}$, then proving $G \models H$ can be substituted by proving $G_1 \models H,\ldots,G_n \models H$. This is because by Clark completion (Section 2.7), G is equivalent to $G_1 \vee \ldots \vee G_n$. However, whether the addition of $G \models H$ to the table $\mathcal{T}$ is sound depends on the use of the set of assumed assertions in the application of AP.

Recall that in the rule AP we require the proof of $(G' \models H') \triangleright (G \models H)$ for $G' \models H'$ an assertion in the table $\mathcal{T}$.

Assume that using our method, given a CLP program Γ, we managed to conclude $G \models H$ where G and H are goals possibly containing atoms and it is not the case that $G \models H$ can be proved without the application of LU (since otherwise trivial by the soundness of RU, DP, CUT, and SPL). Assume that in the proof, there are a number of assumed assertions $A_1,\ldots,A_n$ used coinductively as induction hypotheses in applications of AP. This means that in the proof of $G \models H$ the left unfold rule LU has been applied at least once (possibly interleaved with the applications of of other rules beside AP) obtaining two kinds of assertions:

1. Assertions C which are directly proved using rules other than LU and AP.

2. Assertions B which are proved using AP using some assumed assertion A_j in the table $\mathcal{T}$ as hypothesis for $1 \leq j \leq n$.

We may conclude $\Gamma \Vdash (G \models H)$ holds (cf. Section 2.5.1). From Section 2.6, this is equivalent to $lm(\Gamma) \Rightarrow (G \models H)$. We prove this.

First, define a *refutation* to an assertion $G \models H$ as a successful derivation of one or more atoms in G whose *answer* Ψ has a ground substitution σ such that $\Psi\sigma \wedge H\sigma$ is false (cf. the notion of resolution refutation in Chapter 2). We note that here, an answer Ψ of a goal G consisting of a sequence of atoms $p_1, \ldots, p_m$ and a constraint ϕ is a constraint on the variables of G such that $\Psi\sigma$ holds if and only if $p_i\sigma \in lm(\Gamma)$ for all i in $1 \leq i \leq m$, and $\phi\sigma$ holds.

A finite refutation corresponds to resolution of finite length. A step in the resolution is achieved by left unfold LU rule only. Hence a finite refutation of length k implies a corresponding k left unfold LU applications that result in a contradiction.

$G \models H$ has a finite refutation of length k for some k if and only if $\neg(T_\Gamma \uparrow k \Rightarrow (G \models H))$.

Due to:

1. the soundness of other rules RU, DP, CUT, and SPL, and the partial soundness of LU with the fact that A_i for all $1 \leq i \leq n$ are obtained from $G \models H$ by applying these rules, and

2. all assertions C are proved by RU, CP, CUT and SPL alone,

we have: $G \models H$ holds if A_i holds for all $1 \leq i \leq n$ hold, and these hold if and only if for all i such that $1 \leq i \leq n$, and for all $k \geq 0 : A_i$ has no finite refutation of length k.

We prove inductively:

- **Base case:** When $k = 0$, for all i such that $1 \leq i \leq n$, A_i trivially has no finite refutation of length 0. In other words, for all i, trivially $T_\Gamma \uparrow 0 \equiv \Box \Rightarrow A_i$.

- **Inductive case:** Assume that

  For all i such that $1 \leq i \leq n$, A_i has no finite refutation of length k or less. (5.2)

  we want to prove that

  For all i such that $1 \leq i \leq n$, A_i has no finite refutation of length $k+1$ or less. (5.3)

Notice again in our assumptions above that assertions B are proved by applying AP using A_j for some $1 \leq j \leq n$. Because subsumption holds in every application of AP, this means that for any such B,

$$A_j \rhd B. \tag{5.4}$$

The proof is by contradiction. Now suppose that (5.3) is false, that is, A_i for some i such that $1 \leq i \leq n$ has a finite refutation of length $k+1$ or less. But due to our hypothesis (5.2), A_i has no finite refutation of length k or less. Therefore it must be the case that A_i has a finite refutation of length $k+1$.

Again, note that we have applied LU to A_i at least once, possibly interleaved with applications of RU, DP, CUT, and SPL obtaining the following two kinds of assertions:

1. Assertions C which are proved by applications of RU, DP, CUT, or SPL.
2. Assertions B which are proved by AP using some A_j for $1 \leq j \leq n$ in the table as induction hypothesis.

Then either of these must hold:

1. Some assertion of type C is a refutation to A_i of length $k+1$.
2. Some A_i has a finite refutation of length $k+1$.

For the first case, however, regardless of the length, since all such assertions C are already proved by RU, DP, CUT, and SPL that are sound, this case is not possible.

For the second case, since A_i has to have a finite refutation of length $k+1$, therefore there has to be at least one assertion of type B that is reached in k or less unfolds. Therefore, B has to have a refutation of length k or less. Now since (5.4) holds, then it should be the case that some A_j for $1 \leq j \leq n$ such that A_j also has a finite refutation of length k or less. To see this, here notice that B has a refutation of length k or less, that is, $\neg(T_\Gamma \uparrow k \Rightarrow B)$. The conjunction of this, (5.4), and $T_\Gamma \uparrow k \Rightarrow A_j$ is unsatisfiable. Therefore, by the law of excluded middle it must be the case that $\neg(T_\Gamma \uparrow k \Rightarrow A_j)$, in other words, A_j must have a refutation of length k or less. But this contradicts our hypothesis (5.2) that A_i for all $1 \leq i \leq n$ has no finite refutation of length k or less. $\square$

5.7.2 On Completeness

It is easy to construct an example that demonstrates the incompleteness of our method. Proving $G \models H$ is unsolvable in general [141], even when we assume that we have a perfect constraint solver to solve all interpreted functions and relations.

We argue, however, that our proof method is more complete than proof methods that only consider one level of left unfold before applying inductive reasoning, such as Kanamori and Fujita's [118] and Mesnard et al.'s [141]. Our proof method allows arbitrary level of left unfold and discovers opportunistically the chance to apply inductive reasoning. This approach is also more automatable. The proof process can be considered as an algorithmic search process, where we apply inductive reasoning whenever we encounter a "cycle" or "redundancy" in the proof. We provide a detailed comparison to Mesnard et al.'s proof method in Section 5.10.1 of the appendix.

5.8 Compositional Program Analysis and Verification Framework

There are two major approaches to program reasoning in the literature. The first of these, which is the *abstract interpretation* [35] approach is based on providing abstract description of program states. State-space traversal is then done on abstracted description of reachable states, which is more efficient than concrete (unabstracted) traversal. However, the approach is inherently incomplete due to loss accuracy incurred by the abstraction. This approach is also called *program analysis* approach.

The second approach is originally due to Hoare and Floyd [100]. It is based on composing proofs from proofs of program fragments. Here the correctness of a program fragment is denoted by *triples* of the form $\{\phi\}\ t\ \{\psi\}$ where ϕ and ψ are conditions and t the program fragment. When t and t' appear in sequence in a program, the proofs of $\{\phi\}\ t\ \{\psi\}$ and $\{\phi'\}t'\{\psi'\}$ are then combined to construct the proof of $\{\phi\}tt'\{\psi'\}$, until finally the whole program is proved. This approach is called *program verification*.

Also central to a full-fledged program reasoning framework is compositionality. We may want to verify procedures or program fragments separately in order to simplify the whole proof by avoiding redundant proofs. Program verification is naturally compositional, while program analysis is not. In this section we also introduce compositional reasoning based on our proof rules.

We argue that the difference between abstract interpretation, program verification, and compositional program reasoning is simply the *location* at which abstraction using CUT rule is applied. In traditional abstract interpretation abstraction is applied everywhere while in automated program verification the abstraction is done only at a point within each loop. Finally, in compositional reasoning abstraction is performed at procedure call points or fragment boundaries.

What enables the unifying of program analysis and verification in a single framework is the view of of left unfold (LU) as computation of *strongest postcondition*, which we explain next.

5.8.1 Unfold as Strongest Postcondition Operator

We argue that in forward CLP models (explained in Section 3.1.3), an unfold step corresponds to a strongest postcondition computation. The strongest postcondition of a condition s, denoted $sp(t,s)$ is the smallest set of states to which a transition ρ_t defined by the program fragment t can be taken from any state in s. More formally [19]:

$$sp(t,s) \equiv \langle \exists \tilde{x}_0 : \rho_t(\tilde{x}_0,\tilde{x}) \wedge s\{\tilde{x} \mapsto \tilde{x}_0\}\rangle.$$

The formula $\rho_t(\tilde{x}_0,\tilde{x})$ can be decomposed into a disjunction

$$\rho_1(\tilde{x}_0,\tilde{x}) \vee \ldots \vee \rho_n(\tilde{x}_0,\tilde{x})$$

where each disjunct represents a logical input-output relation induced by an execution path from the start to the end program point of t. Now, the forward CLP program Γ, excluding the constraint fact representing the condition of interest, represents exactly the transition relation ρ_t since it consists of n clauses $\kappa_1,\ldots,\kappa_n$ (excluding constraint fact κ_{n+1}) where clause κ_i, $1 \leq i \leq n$ is:

$$p(\tilde{X}_0) \texttt{ :- } p(\tilde{X}),\rho_i(\tilde{X}_0,\tilde{X}),$$

where $\tilde{X}_0$ and $\tilde{X}$ are the renamings of $\tilde{x}_0$ and $\tilde{x}$ as CLP variables, respectively.

Now,

$$sp(t,s) \equiv \bigvee_{i=1}^{n} \langle \exists \tilde{x}_0 : \rho_i(\tilde{x}_0,\tilde{x}) \wedge s\{\tilde{x} \mapsto \tilde{x}_0\}\rangle.$$

Similarly,

$$unfold_{p(\tilde{X})}(S) = \{S' | \bigvee_{i=1}^{n+1} \Box \texttt{ :- } S' = resolv_{p(\tilde{X})}(\Box \texttt{ :- } S,\kappa_i)\}.$$

The subset of $unfold_{p(\tilde{X})}(S)$ of goals that include p therefore corresponds to $sp(t,s)$. The result of resolution using κ_{n+1} (constraint fact) is a goal S' which does not contain predicate p, hence S' does not represent program states.

Example 5.1. Let us revisit Program 3.1 whose transition relation ρ_{Sum} is given in Example 3.1 (Page 43), and whose forward CLP model is Program 3.3 (Page 49).

Here, $sp(Sum, l = 0 \wedge x = 10 \wedge s = 2) = (l = 1 \wedge x = 10 \wedge s = 2 \wedge n > 10) \vee (l = \Omega \wedge x = 10 \wedge s = 2 \wedge n \leq 10)$. Now, there are two possible resolution steps of the goal $\square$:- $p(0, 10, 2, N)$. (κ_{11}) using the clauses of Program 3.3, which does not result in success ($\square$:- $\square$). The first resolution uses the clause κ_7 :

$$\square \texttt{ :- } p(1, 10, 2, N), N > 10. \quad \kappa_{13} = resolv_{p(0,10,2,N)}(\kappa_{11}, \kappa_7).$$

The second resolution uses the clause κ_8 :

$$\square \texttt{ :- } p(\Omega, 10, 2, N), N \leq 10. \quad \kappa_{14} = resolv_{p(0,10,2,N)}(\kappa_{11}, \kappa_8).$$

Each resolution corresponds to a disjunct in $sp(Sum, l = 0 \wedge x = 10 \wedge s = 2)$.

5.8.2 Intermittent Abstraction

In this section we present a way of engineering in our CLP framework into a general proof method of program reasoning based on abstract interpretation in which the process of abstraction is intermittent, that is, approximation is performed only at selected points in the proof tree, if at all. This is an application of our the CUT rule in Figure 5.3. Here there is no restriction of when abstraction is performed. The key advantages are the following two:

1. The abstract domain required to ensure convergence of the algorithm can be minimized. For example, to reason that $x = 2$ after executing

$$\begin{array}{rcl} x & := & 0 \\ x & := & x+1 \\ x & := & x+1 \end{array}$$

   one needs to know that $x = 1$ holds before the final assignment. Thus, in say a *predicate abstraction* [80] setting, the abstract domain must contain the predicates $x = 0$, $x = 1$ and $x = 2$ for the above reasoning to be possible whereas in our framework it is enough to use one predicate $x = 0$, which holds right after the execution of the first statement. From our discussions in Section 5.8.1, we know that our framework is capable of computing the strongest postcondition of the sequence of $x \ := \ x+1$, which given the input condition $x = 0$ is $x = 2$.

2. Computing the next abstract state in a transition can be expensive. When done naively, abstract transition in a predicate abstraction framework requires exponential calls to theorem prover [9, 80, 189].

We next show how we may perform abstract interpretation, in particular predicate abstraction, possibly intermittently using CUT rule.

Predicate abstraction [80] is a successful method of abstract interpretation [34]. The abstract domain, constructed from a given finite set of predicates over program variables, is intuitive and easily, though not necessarily efficiently, computable within a traversal method of the program's control flow structure. Predicate abstraction has been widely employed jointly with abstraction refinement techniques [97, 11, 10].

In the literature on predicate abstraction, the abstract description is a specialized data structure called *monomials* [41], a.k.a. *cubes* [9]. The abstraction operation serves to propagate a monomial through a small program fragment (a test or a contiguous group of assignments), and then obtaining another monomial. The strength of this method is in the simplicity of using a finite set of predicates over the fixed number of program variables as a basis for the abstract description.

We choose to follow this method. However, our abstract description shall not be a distinguished data structure. In fact, our abstract description of a goal is itself a goal.

Given a finite number of propositions $\varphi_1, \ldots, \varphi_n$, we may abstract a goal $G \equiv p(\tilde{X}), \phi$ into the goal $p(\tilde{X}), \varphi'_1, \ldots, \varphi'_n$ where

$$\varphi'_i \equiv \begin{cases} \varphi_i \text{ when } \phi \Rightarrow \varphi_i \\ \neg\varphi_i \text{ when } \phi \Rightarrow \neg\varphi_i \\ \neg\Box \text{ otherwise} \end{cases}$$

It is obvious that

$$p(\tilde{X}), \phi \models p(\tilde{X}), \varphi'_1, \ldots, \varphi'_n.$$

From an obligation $p(\tilde{X}), \phi \models H$, the CUT rule would produce the two assertions $p(\tilde{X}), \varphi'_1, \ldots, \varphi'_n \models H$, and $p(\tilde{X}), \phi \models p(\tilde{X}), \varphi'_1, \ldots, \varphi'_n$. Since the second assertion produced holds, the CUT rule here in effect strengthens $p(\tilde{X}),\phi \models H$, to $p(\tilde{X}),\varphi'_1,\ldots,\varphi'_n \models H$ by weakening the lhs goal of the assertion.

Example 5.2. The intermittent (predicate) abstraction technique has actually been exemplified in the proof of the Sum program in Section 5.5. We refer to the detailed proof in Figure 5.7. Using CUT we strengthened the assertion 1 into assertion 2 by proving at the side the subsumption 2s.1 and the residual obligation 2r.1. In 2s.1, $\psi \equiv X = 0, S = 0, N \geq 0$, $\varphi_1 \equiv S = (X^2 - X)/2$, $\varphi_2 \equiv X \leq N$, and $\varphi_3 \equiv N \geq 0$. It is important to note that here the abstraction is applied intermittently, that is, only at the assertion whose lhs goal represents the condition at program point $\langle 0 \rangle$.

5.8.3 Program Verification

In this section we demonstrate how our proof method also provides a verification condition computation mechanism in the context of program verification. Program verification was introduced by Hoare in [100], which he attributed to Floyd. In [100] it is qualified as *axiomatic* because of the symbolic treatment of conditions which are constraints depending on the axioms of the underlying theory. In this method, given a sequential program fragment t which is any statement *Stmt* in our simple programming language of Figure 3.1, or any sequential composition of statements called *blocks*, we prove *triples* of the form $\{\phi\}\ t\ \{\psi\}$. The triple says that if the execution starts in a state s where $s \Rightarrow \phi$, then if the execution of t terminates, we reach a state s' where $s' \Rightarrow \psi$. ϕ is called the *precondition* and ψ the *postcondition*. The triple denotes a *partial* correctness of the fragment t since it may still hold independent on whether t actually terminates or not. There is a stronger notion of correctness, which is that of *total* correctness where we also require that t terminates. However, this belongs to the class of liveness properties which is outside the scope of this thesis.

The program verification approach includes a number of proof rules to infer the triples. For our simple programming language they are shown in Figure 5.12. Program verification only handles structured program such that we exclude the consideration for **goto**s.

Being able to perform program verification distinguishes our framework from normal abstract interpretation. In a normal abstract interpretation framework, it is not easy to provide a simple abstraction that would be equivalent or close approximation of loop invariants. This is because the abstraction applies at every condition or state of a program. In contrast, in our framework, abstraction can be applied intermittently (Section 5.8.2) such that we can apply loop invariants

$$\text{(COMPOSITION)}\quad \frac{\{\phi\}\ t_1\ \{\eta\} \quad \{\eta\}\ t_2\ \{\psi\}}{\{\phi\}\ t_1\ t_2\ \{\psi\}}$$

$$\text{(ASSIGNMENT)}\quad \frac{}{\{\psi\{x \mapsto E\}\}\ x\ :=\ E\ \{\psi\}}$$

$$\text{(SKIP)}\quad \frac{}{\{\phi\}\ \textbf{skip}\ \{\phi\}}$$

$$\text{(IF}_1\text{)}\quad \frac{\{\phi \wedge \beta\}\ t\ \{\psi\} \quad \phi \wedge \neg\beta \Rightarrow \psi}{\{\phi\}\ \textbf{if}\ (\beta)\ \textbf{then}\ t\ \textbf{end if}\ \{\psi\}}$$

$$\text{(IF}_2\text{)}\quad \frac{\{\phi \wedge \beta\}\ t_1\ \{\psi\} \quad \{\phi \wedge \neg\beta\}\ t_2\ \{\psi\}}{\{\phi\}\ \textbf{if}\ (\beta)\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \textbf{end if}\ \{\psi\}}$$

$$\text{(WHILE)}\quad \frac{\{\phi \wedge \beta\}\ t\ \{\phi\}}{\{\phi\}\ \textbf{while}\ (\beta)\ \textbf{do}\ t\ \textbf{end do}\ \{\phi \wedge \neg\beta\}}$$

$$\text{(IMPLIED)}\quad \frac{\phi' \Rightarrow \phi \quad \{\phi\}\ t\ \{\psi\} \quad \psi \Rightarrow \psi'}{\{\phi'\}\ t\ \{\psi'\}}$$

Figure 5.12: Program Verification Proof Rules

to generalize the context of a loop at the program point at which the loop is located. When the user provides all the loop invariants necessary for the program, the proof process terminates automatically.

We now discuss how we may accommodate program verification in our framework. We assume that a sequential program P has been translated into CLP, and we want to verify $\{\phi_P\}\ P\ \{\psi_P\}$. Here we consider proving $\{\phi\}\ t\ \{\psi\}$ compositionally, under various cases of t, where t is a fragment of P. We make use of LU, CUT, and AP rules.

Case t is:

- **A sequential composition** $t_1\ t_2$. According to the COMPOSITION rule (Figure 5.12), we provide a condition η and we prove separately $\{\phi\}\ t_1\ \{\eta\}$ and $\{\eta\}\ t_2\ \{\psi\}$.

  Suppose that t_1 starts at program point $\langle l \rangle$, t_2 at $\langle l' \rangle$, and the program point right after t_2 is $\langle l'' \rangle$. We start with an obligation

$$p(l,\tilde{X},\tilde{X}_f),\phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  We apply our proof rules according to the statements in t_1, stopping upon producing obligations of the form

$$p(l',\tilde{X},\tilde{X}_f),\eta(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

This is a proof of $\{\phi\}\ t_1\ \{\eta\}$.

We then start with applying our rules to the last assertion, stopping upon producing obligations of the form

$$p(l',\tilde{X},\tilde{X}_f),\psi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

Here we claim that we have established $\{\eta\}\ t_2\ \{\psi\}$.

- **An assignment.** Suppose that we have n program variables $x_1,\ldots,x_n$ and we want to prove $\{\phi\}\ t\ \{\psi\}$. where t is the sequence the single statement $x_i\ := \ f(\tilde{x})$ for some i such that $1 \le i \le n$. Here t_1 represents the transition relation $\tau(\tilde{x},\tilde{x}') \equiv x_i' = f(\tilde{x}) \wedge \bigwedge_{j \neq i} x_j' = x_j$. By a translation of the program P into a forward CLP model with final variables, we have the following CLP clause:

$$p(l,\tilde{X},\tilde{X}_f)\ \texttt{:-}\ \tau(\tilde{X},\tilde{X}'),p(\mathit{next\_label}(l),\tilde{X}',\tilde{X}_f).$$

Assuming t is located at program point $\langle l \rangle$, here we again start with the obligation

$$p(l,\tilde{X},\tilde{X}_f),\phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

We apply LU to this obligation using the above CLP clause obtaining the obligation

$$p(\mathit{next\_label}(l),\tilde{X}',\tilde{X}_f),\tau(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

Here what we have done is a strongest postcondition propagation establishing $\{\phi\}\ t\ \{sp(t,\phi)\}$. Since $sp(t,\phi)\{x_i \mapsto f(\tilde{x})\}$ is equivalent to ϕ, this corresponds to the use of the ASSIGNMENT axiom of Figure 5.12.

In case $sp(t,\phi)$ is not (trivially) equivalent to ψ, we still need to use IMPLICATION rule of Figure 5.12, to establish $\{\phi\}\ t\ \{\psi\}$ by proving, as an obligation, $sp(t,\phi) \Rightarrow \psi$. Here we apply CUT rule to the last obligation above obtaining

$$p(\mathit{next\_label}(l),\tilde{X}',\tilde{X}_f),\psi(\tilde{X}') \models \psi_P(\tilde{X}_f).$$

The side condition of the CUT rule requires us to prove the subsumption

$$p(next\_label(l),\tilde{X}',\tilde{X}_f),\tau(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models p(next\_label(l),\tilde{X}',\tilde{X}_f),\psi(\tilde{X}').$$

We prove this in a special way by proving

$$\tau(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \Rightarrow \psi(\tilde{X}'),$$

which is indeed the very proof of $sp(t,\phi) \Rightarrow \psi$. Other than the subsumption, we also prove the trivial residual obligation $\psi_P(\tilde{X}_f) \models \psi_P(\tilde{X}_f)$.

- **A skip.** In case t is **skip**, it represents the transition relation $\tau(\tilde{x},\tilde{x}') \equiv \tilde{x} = \tilde{x}'$. Assuming the statement is located at program point $\langle l \rangle$, we have in the forward CLP model the clause

  $$p(l,\tilde{X},\tilde{X}_f) \ \texttt{:-}\ \ p(next\_label(l),\tilde{X},\tilde{X}_f).$$

  We apply the LU rule using the above clause to the obligation

  $$p(l,\tilde{X},\tilde{X}_f),\phi(\tilde{X}) \models \psi_P(\tilde{X}_f)$$

  obtaining the new obligation

  $$p(next\_label(l),\tilde{X},\tilde{X}_f),\phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  This process is equivalent to the application of the SKIP axiom of Figure 5.12.

  In case ϕ is not trivially equivalent to ψ, we need to apply the IMPLICATION rule of Figure 5.12 to prove $\phi \Rightarrow \psi$. This is done in our framework by applying the CUT rule to the last obligation obtaining the new obligation

  $$p(next\_label(l),\tilde{X},\tilde{X}_f),\psi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  The application of CUT requires us to prove the subsumption

  $$p(next\_label(l),\tilde{X},\tilde{X}_f),\phi(\tilde{X}) \models p(next\_label(l),\tilde{X},\tilde{X}_f),\psi(\tilde{X}).$$

This we may prove by instead proving the sufficient condition $\phi(\tilde{X}) \Rightarrow \psi(\tilde{X})$, which is exactly the proving of $\phi \Rightarrow \psi$ mentioned above. The residual obligation here is trivial.

- **An if conditional without else part.** In case t is **if** (β) **then** t_1 **end if** , according to the IF$_1$ rule of Figure 5.12, we need to prove separately $\{\phi \wedge \beta\}\ t_1\ \{\eta\}$ and $\phi \wedge \neg\beta \Rightarrow \eta$.

  Assuming t starts at program point $\langle l \rangle$, we have in our CLP model of P the clauses

$$\begin{aligned} p(l,\tilde{X},\tilde{X}_f) &\ \texttt{:-}\ \beta(\tilde{X}), p(next\_label\_then(l),\tilde{X},\tilde{X}_f). \\ p(l,\tilde{X},\tilde{X}_f) &\ \texttt{:-}\ \neg\beta(\tilde{X}), p(next\_label(l),\tilde{X},\tilde{X}_f). \end{aligned}$$

  We start the verification again with the obligation

$$p(l,\tilde{X},\tilde{X}_f), \phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  We apply a complete left unfold LU using the two clauses of CLP above resulting in the following two obligations:

$$\begin{aligned} &p(next\_label\_then(l),\tilde{X},\tilde{X}_f), \phi(\tilde{X}), \beta(\tilde{X}) \models \psi_P(\tilde{X}_f). \\ &p(next\_label(l),\tilde{X},\tilde{X}_f), \phi(\tilde{X}), \neg\beta(\tilde{X}) \models \psi_P(\tilde{X}_f). \end{aligned}$$

  Here we assume that we proved $\{\phi \wedge \beta\}\ t_1\ \{\psi\}$ using our rules, which means that there is an application of our proof rules which transforms the first obligation into

$$p(next\_label(l),\tilde{X},\tilde{X}_f), \psi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  For the second obligation above, we apply CUT rule obtaining the same obligation. The application of CUT rule requires us to prove the subsumption

$$p(next\_label(l),\tilde{X},\tilde{X}_f), \phi(\tilde{X}), \neg\beta(\tilde{X}) \models p(next\_label(l),\tilde{X},\tilde{X}_f), \psi(\tilde{X}).$$

  This we establish by proving the sufficient condition

$$\phi(\tilde{X}), \neg\beta(\tilde{X}) \Rightarrow \psi(\tilde{X}),$$

  which indeed is the proof of $\phi \wedge \neg\beta \Rightarrow \psi$. The residual obligation here is again trivial.

- **An if conditional with else part.** In case t is **if** (β) **then** t_1 **else** t_2 **end if** then according to the IF$_2$ rule of Figure 5.12 we replace the obligation with the proof of both the obligations $\{\phi \wedge \beta\}\ t_1\ \{\psi\}$ and $\{\phi \wedge \neg\beta\}\ t_2\ \{\psi\}$.

  Again, assuming t starts at program point $\langle l \rangle$, we have in our CLP model of P the clauses

$$\begin{aligned} p(l,\tilde{X},\tilde{X}_f) &\ \texttt{:-}\ \beta(\tilde{X}), p(\mathit{next\_label\_then}(l),\tilde{X},\tilde{X}_f). \\ p(l,\tilde{X},\tilde{X}_f) &\ \texttt{:-}\ \neg\beta(\tilde{X}), p(\mathit{next\_label\_else}(l),\tilde{X},\tilde{X}_f). \end{aligned}$$

  We start the verification again with the obligation

$$p(l,\tilde{X},\tilde{X}_f), \phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  We apply a complete left unfold LU using the two clauses of CLP above resulting in the following two obligations:

$$\begin{aligned} &p(\mathit{next\_label\_then}(l),\tilde{X},\tilde{X}_f), \phi(\tilde{X}), \beta(\tilde{X}) \models \psi_P(\tilde{X}_f). \\ &p(\mathit{next\_label\_else}(l),\tilde{X},\tilde{X}_f), \phi(\tilde{X}), \neg\beta(\tilde{X}) \models \psi_P(\tilde{X}_f). \end{aligned}$$

  Here we assume that we proved both $\{\phi \wedge \beta\}\ t_1\ \{\psi\}$ and $\{\phi \wedge \neg\beta\}\ t_2\ \{\psi\}$ using our rules, which means that there is an application of our proof rules which transforms the both obligations into the single obligation

$$p(\mathit{next\_label}(l),\tilde{X},\tilde{X}_f), \psi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

  Hence we have applied program verification rules to prove $\{\phi\}\ t\ \{\psi\}$.

- **A while loop.** Say t is of the syntax **while** (β) **do** t_1 **end do** . We therefore want to establish $\{\phi\}$ **while** (β) **do** t_1 **end do** $\{\psi\}$. Here we require a *loop invariant*, which is ϕ itself in the WHILE rule of Figure 5.12. However, in general we may fail to prove $\{\phi \wedge \beta\}\ t_1\ \{\phi\}$. We therefore allow the user to manually provide a loop invariant ξ where $\phi \Rightarrow \xi$.

  To provide a loop invariant, we require the application of IMPLICATION rule, such that we decompose the original obligation into the three obligations

  1. $\phi \Rightarrow \xi$,
  2. $\{\xi\}\ t\ \{\xi \wedge \neg\beta\}$, and

3. $\xi \wedge \neg\beta \Rightarrow \psi$.

Again we assume that t is located at program point $\langle l \rangle$. In our framework, we start the proof with an obligation

$$p(l, \tilde{X}, \tilde{X}_f), \phi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

We first apply CUT rule to this condition to obtain the new obligation

$$p(l, \tilde{X}, \tilde{X}_f), \xi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

Here we prove at the side $\phi(\tilde{X}) \Rightarrow \xi(\tilde{X})$ which is the sufficient condition for subsumption, establishing verification obligation 1 above. The residual obligation for this application of CUT is trivial: $\psi_P(\tilde{X}_f) \models \psi_P(\tilde{X}_f)$.

By the translation of P into a forward CLP model, we have the following CLP clauses:

$$\begin{array}{rcl} p(l, \tilde{X}, \tilde{X}_f) & :- & \beta(\tilde{X}), p(next\_label\_then(l), \tilde{X}, \tilde{X}_f). \\ p(l, \tilde{X}, \tilde{X}_f) & :- & \neg\beta(\tilde{X}), p(next\_label(l), \tilde{X}, \tilde{X}_f). \end{array}$$

We then unfold our last obligation completely using these clauses obtaining the obligations

a. $p(next\_label\_then(l), \tilde{X}, \tilde{X}_f), \xi(\tilde{X}), \beta(\tilde{X}) \models \psi_P(\tilde{X}_f)$.

b. $p(next\_label(l), \tilde{X}, \tilde{X}_f), \xi(\tilde{X}), \neg\beta(\tilde{X}) \models \psi_P(\tilde{X}_f)$.

It is important to note that here, since we have applied a left unfold, the “ancestor” assertion $p(l, \tilde{X}, \tilde{X}_f), \xi(\tilde{X}) \models \psi_P(\tilde{X}_f)$ is now kept in the table.

Further, we assume that we proved $\{\xi \wedge \beta\}\ t_1\ \{\xi\}$ by IMPLICATION rule, where we proved separately $\{\xi \wedge \beta\}\ t_1\ \{\alpha\}$ and $\alpha \Rightarrow \xi$ for some α. Here we assume that we have proved $\{\xi \wedge \beta\}\ t_1\ \{\alpha\}$.

Also since t_1 is a loop body, the next program point of its last statement is $\langle l \rangle$. From these, we may replace the obligation (a) above with

$$p(l, \tilde{X}, \tilde{X}_f), \alpha(\tilde{X}) \models \psi_P(\tilde{X}_f). \tag{5.5}$$

By the previous application of LU, here the table contains the assertion $p(l, \tilde{X}, \tilde{X}_f), \xi(\tilde{X}) \models \psi_P(\tilde{X}_f)$. We use this assertion to prove (5.5) by an application of AP. In this proof, we are required to prove both the subsumption and the residual obligation. The residual obligation in this case is $\psi_P(\tilde{X}_f) \models \psi_P(\tilde{X}_f)$ and is trivial. The subsumption here is the obligation

$$p(l,\tilde{X},\tilde{X}_f),\alpha(\tilde{X}) \models p(l,\tilde{X},\tilde{X}_f),\xi(\tilde{X}).$$

This holds when we prove the sufficient condition $\alpha(\tilde{X}) \Rightarrow \xi(\tilde{X})$, which is one of the premises of the IMPLICATION rule mentioned above.

For obligation (b), we apply the CUT rule to it obtaining the new obligation

$$p(\mathit{next\_label}(l),\tilde{X},\tilde{X}_f),\psi(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

Here we prove the sufficient condition $\xi(\tilde{X}) \wedge \neg\beta(\tilde{X}) \Rightarrow \psi(\tilde{X})$ of the subsumption, which is the proof of the program verification obligation no. 3 above. Again the residual obligation here is $\psi_P(\tilde{X}_f) \models \psi_P(\tilde{X}_f)$, which is trivial.

We note that in our framework, when proving P using program verification technique we would eventually encounter obligations of the form

$$p(\Omega,\tilde{X},\tilde{X}_f),\psi_P(\tilde{X}) \models \psi_P(\tilde{X}_f).$$

that is, when the left unfolds reach the end of the program. By our translation of P into forward CLP model, we have the constraint fact

$$p(\Omega,\tilde{X},\tilde{X}).$$

Unfolding using this fact results in the obligation $\psi_P(\tilde{X}) \models \psi_P(\tilde{X})$ which immediately holds.

Note that what we require as ingredients of a verification system with automated condition generation based on strongest postcondition are the rules LU, AP, and applications of CUT at fragment boundaries treated in Figure 5.12. Both the intermittent abstraction approach (Section 5.8.2) and program verification approach presented here are accommodated by our overall algorithm to be presented in Chapter 6. The key is the specification of abstraction points by the user at which to apply the CUT rule.

Example 5.3. We apply program verification technique to the verification of the Sum problem in Example 5.1 (Page 131). A proof using our rules has been provided in Figure 5.7 (Page 122). The proof can be considered to have used intermittent abstraction (Example 5.2 on Page 5.2). Here we argue that it is also a program verification proof.

Notice that here we prove a while loop:

```
      {$x = s = 0, n \geq 0$}
⟨0⟩   while ($x < n$) do
⟨1⟩     $s$ := $s + x$
⟨2⟩     $x$ := $x + 1$
      end do
      {$s = (n^2 - n)/2$}
```

Using the IMPLICATION rule we prove $x = s = 0, n \geq 0 \Rightarrow s = (x^2 - x)/2, x \leq n, n \geq 0$ and

```
      {$s = (x^2 - x)/2, x \leq n, n \geq 0$}
⟨0⟩   while ($x < n$) do
⟨1⟩     $s$ := $s + x$
⟨2⟩     $x$ := $x + 1$
      end do
      {$s = (n^2 - n)/2$}
```

This corresponds to the application of CUT to obligation 1 which produces obligation 2 in Figure 5.7.

We now apply the WHILE rule of program verification obtaining the two obligations

```
      {$s = (x^2 - x)/2, x < n, n \geq 0$}
⟨1⟩     $s$ := $s + x$
⟨2⟩     $x$ := $x + 1$                          (5.6)
      {$s = (x^2 - x)/2, x \leq n, n \geq 0$}
```

and

$$s = (x^2 - x)/2, x = n, n \geq 0 \Rightarrow s = (n^2 - n)/2. \tag{5.7}$$

The second obligation obviously holds, and this corresponds to the proof of obligation 4 in Figure

5.7. Note that here we deviate from what has been suggested to prove a while conditional in that we do not use the CUT rule. Nevertheless we still prove the same obligation (5.7).

To prove (5.6) we perform strongest postcondition computation across the two statements. According to our program verification technique, here we prove that

$$\begin{array}{ll} & \{s = (x^2 - x)/2, x < n, n \geq 0\} \\ \langle 1 \rangle & \quad s \ := \ s + x \\ \langle 2 \rangle & \quad x \ := \ x + 1 \\ & \{\alpha\} \end{array}$$

and $\alpha \Rightarrow s = (x^2 - x)/2, x \leq n, n \geq 0$ for some α. Here α is the strongest postcondition across the two statements, which is $s = (x^2 - x)/2, x \leq n, n \geq 0$. Hence $\alpha \Rightarrow s = (x^2 - x)/2, x \leq n, n \geq 0$ is immediate.

In Figure 5.7 we perform two left unfold steps from 3b to 6, and then from 6 to 7, which correspond to the strongest postcondition computation across the two statements (note that unfolding corresponds to strongest postcondition, as has been discussed in Section 5.8.1). We then apply the AP rule whose subsumption test establishes $\alpha \Rightarrow s = (x^2 - x)/2, x \leq n, n \geq 0$.

5.8.4 Compositional Program Reasoning

Our CUT rule also allows us to perform compositional verification. Here we can prove program fragments or procedures separately and combine the verification results at the end.

Let us now verify the multiprocedure Program 3.12 with its CLP model Program 3.13. According to [179] (from which the example is taken) here we want to demonstrate that at program point $\langle 2 \rangle$ in the *main* procedure, the assignment $t := a \times b$ is not necessary since at $\langle 2 \rangle$ the relation $t = a \times b$ always holds. The property that at $\langle 2 \rangle$ in *main* the relation $t = a \times b$ can be expressed as the assertion

$$main(0, T, A, B, T_f, A_f, B_f) \models T_f = A_f \times B_f. \tag{5.8}$$

To prove the above assertion, there are two methods that we can use. The first, non-compositional method is to apply LU as usual until we can establish an assertion either via application other rules (mainly DP or AP). The second alternative is to prove the assertion compositionally since the program has a compositional structure. This is done by first by proving an assertion on the procedure p and then using this assertion we prove (5.8).

1	$p(0,T,A,B,T_f,A_f,B_f), T = A \times B \models T_f = A_f \times B_f$	
2a	$p(1,T,A,B,T_f,A_f,B_f), T = A \times B, A = 0 \models T_f = A_f \times B_f$	LU 1
2b	$p(2,T,A,B,T_f,A_f,B_f), T = A \times B, A \neq 0 \models T_f = A_f \times B_f$	LU 1
3	$p(\Omega,T,A,B,T_f,A_f,B_f), T = A \times B, A = 0 \models T_f = A_f \times B_f$	LU 2a
4	$T_f = A_f \times B_f, A_f = 0 \models T_f = A_f \times B_f$	LU 3
5	$\neg\Box$	DP 4
6	$p(3,T,A-1,B,T_f,A_f,B_f), T = A \times B, A \neq 0 \models T_f = A_f \times B_f$	LU 2b
7	$p(0,T,A-1,B,T',A',B'), p(4,T',A',B',T_f,A_f,B_f), T = A \times B, A \neq 0$ $\models T_f = A_f \times B_f$	LU 6
8	$p(0,T,A-1,B,T',A',B'), p(\Omega,T'',A',B',T_f,A_f,B_f), T'' = A' \times B',$ $T = A \times B, A \neq 0 \models T_f = A_f \times B_f$	LU 7
9	$p(0,T,A-1,B,T',A_f,B_f), T_f = A_f \times B_f, T = A \times B, A \neq 0 \models T_f = A_f \times B_f$	LU 8
10	$\neg\Box$	DP 9

1	$main(0,T,A,B,T_f,A_f,B_f) \models T_f = A_f \times B_f$	
2	$main(1,T',A,B,T_f,A_f,B_f), T' = A \times B \models T_f = A_f \times B_f$	LU 1
3	$p(0,T',A,B,T'',A',B'), main(2,T'',A',B',T_f,A_f,B_f), T' = A \times B$ $\models T_f = A_f \times B_f$	LU 2
4	$main(2,T'',A',B',T_f,A_f,B_f), T'' = A' \times B' \models T_f = A_f \times B_f$	CUT 3
5	$T_f = A_f \times B_f \models T_f = A_f \times B_f$	LU 4
6	$\neg\Box$	DP 5

Figure 5.13: Compositional Proof of Sharir-Pnueli's Example

Here we demonstrate the compositional proof. We first prove the following assertion on the procedure p :

$$p(0,T,A,B,T_f,A_f,B_f)T = A \times B \models T_f = A_f \times B_f. \tag{5.9}$$

We use this assertion in the proof of (5.8).

The complete compositional proof of (5.8) is shown in Figure 5.13. In assertion (3) in the proof of (5.8), we use the assertion (5.9) to establish the validity of the CUT application.

Compositional proof is not applicable only to multiprocedure programs. In a normal programs, we may want to prove program fragments separately. In explaining this, we introduce again a new example Program 5.2, whose forward CLP model is Program 5.3. We can imagine this program to be divided into two fragments: The first fragment consists of statements from $\langle 0 \rangle$ to $\langle 3 \rangle$, and the second fragment consists of statements from $\langle 4 \rangle$ to Ω.

For the proof of the whole program, we may prove each fragments separately. This compositional proof is shown in Figure 5.14, where we prove

$$p(0,X,A,B,C,X_f), X > 0 \models X_f > 0. \tag{5.10}$$

```
⟨0⟩   if ($a = 1$) then
⟨1⟩     skip
      end if
⟨2⟩   if ($b = 1$) then
⟨3⟩     $c$ := 0
      end if
⟨4⟩   if ($c = 1$) then
⟨5⟩     $x$ := $x + 1$
      end if
```

Program 5.2: Simple If Sequence Program

$$\begin{array}{lll}
p(0,X,A,B,C,X_f) & \text{:-} & p(1,X,A,B,C,X_f), A = 1. \\
p(0,X,A,B,C,X_f) & \text{:-} & p(2,X,A,B,C,X_f), A \neq 1. \\
p(1,X,A,B,C,X_f) & \text{:-} & p(2,X,A,B,C,X_f). \\
p(2,X,A,B,C,X_f) & \text{:-} & p(3,X,A,B,C,X_f), B = 1. \\
p(2,X,A,B,C,X_f) & \text{:-} & p(4,X,A,B,C,X_f), B \neq 1. \\
p(3,X,A,B,C,X_f) & \text{:-} & p(4,X,A,B,0,X_f). \\
p(4,X,A,B,C,X_f) & \text{:-} & p(5,X,A,B,C,X_f), C = 1. \\
p(4,X,A,B,C,X_f) & \text{:-} & p(\Omega,X,A,B,C,X_f), C \neq 1. \\
p(5,X,A,B,C,X_f) & \text{:-} & p(\Omega,X+1,A,B,C,X_f). \\
p(\Omega,X,A,B,C,X). & &
\end{array}$$

Program 5.3: Simple If Sequence Program CLP Model

In Figure 5.14, we first establish

$$p(4,X,A,B,C,X_f), X > 0 \models X_f > 0.$$

This we use in the proof of (5.10) to establish the validity of the CUT applications. It is easy to see that non-compositional proofs would be larger since assertions (5), (4b), (11), and (10b) in the proof of (5.10) would have been expanded into subtrees.

We can explain on why the proof becomes smaller compositionally, if we see our compositional proof here as performing an intermittent abstraction at program point ⟨4⟩. Whenever ⟨4⟩ is visited in the proof of (5.10) in Figure 5.14, we abstract (5), (4b), (11), and (10b) using CUT into $p(4,X,A,B,C,X_f), X > 0 \models X_f > 0$, hence the three assertions (4b), (11), and (10b) are just redundant to (5).

1	$p(4,X,A,B,C,X_f),X>0 \models X_f>0$	
2a	$p(5,X,A,B,C,X_f),C=1,X>0 \models X_f>0$	LU 1
2b	$p(\Omega,X,A,B,C,X_f),C=0,X>0 \models X_f>0$	LU 1
3	$p(\Omega,X+1,A,B,C,X_f),C=1,X>0 \models X_f>0$	LU 2a
4	$X_f=X+1,C=1,X>0 \models X_f>0$	LU 3
5	$\neg\Box$	DP 4
6	$X=X_f,C=0,X>0 \models X_f>0$	LU 2b
7	$\neg\Box$	DP 6

1	$p(0,X,A,B,C,X_f),X>0 \models X_f>0$	
2a	$p(1,X,A,B,C,X_f),A=1,X>0 \models X_f>0$	LU 1
2b	$p(2,X,A,B,C,X_f),A=0,X>0 \models X_f>0$	LU 1
3	$p(2,X,A,B,C,X_f),A=1,X>0 \models X_f>0$	LU 2a
4a	$p(3,X,A,B,C,X_f),A=1,B=1,X>0 \models X_f>0$	LU 3
4b	$p(4,X,A,B,C,X_f),A=1,B=0,X>0 \models X_f>0$	LU 3
5	$p(4,X,A,B,0,X_f),A=1,B=1,X>0 \models X_f>0$	LU 4a
6	$X_f>0,A=1,B=1,X>0 \models X_f>0$	CUT 5
7	$\neg\Box$	DP 6
8	$X_f>0,A=1,B=0,X>0 \models X_f>0$	CUT 4b
9	$\neg\Box$	DP 8
10a	$p(3,X,A,B,C,X_f),A=0,B=1,X>0 \models X_f>0$	LU 2b
10b	$p(4,X,A,B,C,X_f),A=0,B=0,X>0 \models X_f>0$	LU 2b
11	$p(4,X,A,B,0,X_f),A=0,B=1,X>0 \models X_f>0$	LU 10a
12	$X_f>0,A=0,B=1,X>0 \models X_f>0$	CUT 11
13	$\neg\Box$	DP 12
14	$X_f>0,A=0,B=0,X>0 \models X_f>0$	CUT 10b
15	$\neg\Box$	DP 14

Figure 5.14: Compositional Proof of Simple If Sequence Program

5.9 Verification of Recursive Data Structures

As discussed in Chapter 4, our assertion $G \models H$ allows G and H to include any predicate defined in a CLP program. Here we deal with how we may prove assertions stating properties concerning recursive data structures, which we have presented in Section 4.4.

5.9.1 Proving Basic Constraints

In this paper, we assume the existence of a constraint solver which can reason about integer constraints. Recall however, that we also have array elements as integer terms, and so we describe here a straightforward method of translating an integer constraint containing array expressions into an equivalent one that does not.

Suppose the goal G at hand contains an array element with a composite array expression, say

$\langle H,I,J\rangle[K]$. We then rewrite G into

$$\begin{aligned} &G\langle H,I,J\rangle[K] \mapsto J \text{ when } G \Rightarrow I = K, \text{ and} \\ &G\langle H,I,J\rangle[K] \mapsto H[K] \text{ when } G \Rightarrow I \neq K \end{aligned} \tag{5.11}$$

These rules are due to McCarthy's array axioms [139]. In some cases, we cannot determine whether $G \Rightarrow I = K$ or $G \Rightarrow I \neq K$, in which case we leave the expression $\langle H,I,J\rangle[K]$ in G as is. Whenever $\langle H,I,J\rangle[K]$ is rewritten into $H[K]$, further it can be treated as regular integer variable.

Another useful rule when proving a goal G containing array expression is

$$(G \Rightarrow I = J) \Rightarrow H[I] = H[J]. \tag{5.12}$$

Of course, even in this case we may not always know whether $G \Rightarrow I = J$. We call both simplifications of (5.11) and (5.12) as *array index principle* (*AIP*) simplification.

At first, it seems hopeless to be able to reason about goals containing array update and references efficiently. Fortunately, in most cases, whenever $\langle H,I,J\rangle[K]$ is encountered, it is known whether $I = K$ or $I \neq K$, and usually whenever two distinct expressions $H[I]$ and $H[J]$ are encountered, it is known whether $I = J$ or $I \neq J$.

We now present another inference rule which, though not formally required, is very useful in practice. The idea is that when an assertion predicate describes a heap and one or more constituent data structures, that changes to the heap outside the reachable cells of the data structures are irrelevant.

Suppose the assertion predicate at hand is of the form: $a(H,X)$, where X is the address of a root node (e.g., head of a linked list) of a data structure on a heap H. The *separation principle* (*SEP*) states that

$$a(\langle H,I,J\rangle,X) \equiv a(H,X)$$

when $no\_share(H,X,I)$ holds. Recall from Section 4.4 that $no\_share(H,X,I)$ declares the separation of the data structure rooted at X and I.

This principle, while clearly cannot be a priori guaranteed for an arbitrary user-defined predicate a, generally holds in most cases. In fact, we discover its instances for all data structure verification examples that we have considered so far. The practical use of this principle is to immediately simplify array expressions $\langle H,I,J\rangle$ into H. The use of this principle can sometimes be avoided, such as in the proof in the next section.

1	$p(0,H,P,H_f,P_f), P \neq 0, P = P_0 \models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	
2	$p(1,H,P,H_f,P_f), P \neq 0, P = P_0$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 1
3	$p(2,H',P,H_f,P_f), P \neq 0, P = P_0, H' = \langle H,P,0 \rangle$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 2
4	$p(2,H',P,H_f,P_f), allz(H,P_0,P,H')$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	CUT 3
5	$p(0,H',P',H_f,P_f), allz(H,P_0,P,H'), P' = H'[P+1]$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 4
6a	$p(\Omega,H',P',H_f,P_f), allz(H,P_0,P,H'), P' = H'[P+1], P' = 0$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 5
6b	$p(1,H',P',H_f,P_f), allz(H,P_0,P,H'), P' = H'[P+1], P' \neq 0$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 5
7	$H' = H_f, P' = P_f, allz(H,P_0,P,H'), P' = H'[P+1], P' = 0$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 6a
8	$\neg\Box$	DP 7 $\{Last \mapsto P\}$
9	$p(2,H'',P',H_f,P_f), allz(H,P_0,P,H'), P' = H'[P+1], P' \neq 0,$	
	$H'' = \langle H',P',0 \rangle$	
	$\models allz(H,P_0,?Last,H_f), P_f = H_f[?Last+1], P_f = 0$	LU 6b
10	$\neg\Box$	AP 4,9

Figure 5.15: Proof of List Reset Program

5.9.2 Handling Different Recursions: Linked List Reset

Now let us re-visit Program 3.7 and its CLP model Program 3.8 discussed in Section 3.1.6, We now prove assertion (4.2) we give in Section 4.4:

$$p(0,H,P,H_f,P_f), P \neq 0 \models allz(H,P,?Last,H_f), H[?Last+1] = P_f, P_f = 0, \tag{5.13}$$

using the definition of *allz* (4.3). The assertion states that at the end of the program's execution, the list has been converted to one whose values have been assigned 0.

We give the proof of Program 3.7 in Figure 5.15. We apply AP at 10, which requires the proof of subsumption which is provided in Figure 5.16 and the proof of the residual obligation in Figure 5.17. The proof in Figure 5.15 has a similar structure with the proof of the Sum program in Section 5.5. The main difference being the use of recursive predicates, in this case *allz*, in the assertions.

We note that the assertion (5.13) can be equivalently written as

$$p(0,H,P,H_f,P_f), P \neq 0, P = P_0 \models allz(H,P_0,?Last,H_f), H[?Last+1] = P_f, P_f = 0,$$

10s.1	$allz(H,P_0,P,H'),P'=H'[P+1],P'\neq 0,H''=\langle H',P',0\rangle$	
	$\models allz(H,P_0,P',H'')$	
10s.1′	$allz(H,P_0,P,H'),H'[P+1]\neq 0$	
	$\models allz(H,P_0,H'[P+1],\langle H',H'[P+1],0\rangle)$	Simplified 10s.1
10s.2a	$allz(H,H[P_0+1],P,H_1),H'=\langle H_1,P_0,0\rangle,P_0\neq 0,H'[P+1]\neq 0$	
	$\models allz(H,P_0,H'[P+1],\langle H',H'[P+1],0\rangle)$	LU 10s.1′
10s.2b	$H'=\langle H,P_0,0\rangle,P_0=P,P_0\neq 0,H'[P+1]\neq 0$	
	$\models allz(H,P_0,H'[P+1],\langle H',H'[P+1],0\rangle)$	LU 10s.1′
10s.2a′	$allz(H,H[P_0+1],P,H_1),H_1[P+1]\neq 0,P_0\neq 0$	
	$\models allz(H,P_0,H_1[P+1],\langle\langle H_1,P,0\rangle,H_1[P+1],0\rangle)$	Simplified 10s.2a
10s.3	$\neg\Box$	AP 10s.1′, 10s.2a′
10s.2b′	$P_0\neq 0,H[P_0+1]\neq 0$	
	$\models allz(H,P_0,H[P_0+1],\langle\langle H,P_0,0\rangle,H[P_0+1],0\rangle)$	Simplified 10s.2b
10s.4	$P_0\neq 0,H[P_0+1]\neq 0$	
	$\models allz(H,H[P_0+1],H[P_0+1],?H_1),P_0\neq 0,$	
	$\langle ?H_1,P_0,0\rangle=\langle\langle H,P_0,0\rangle,H[P_0+1],0\rangle$	RU 10s.2b′
10s.5	$P_0\neq 0,H[P_0+1]\neq 0$	
	$\models P_0\neq 0,H[P_0+1]\neq 0,$	
	$\langle\langle H,H[P_0+1],0\rangle,P_0,0\rangle=\langle\langle H,P_0,0\rangle,H[P_0+1],0\rangle$	RU 10s.4
10s.6	$\neg\Box$	DP 10s.5

10s.3s.1	$allz(H,H[P_0+1],P,H_1),H_1[P+1]\neq 0,P_0\neq 0$	
	$\models allz(H,H[P_0+1],P,H_1),H_1[P+1]\neq 0$	AP 10s.1′, 10s.2a′
10s.3s.2	$\neg\Box$	DP 10s.3s.1

10s.3r.1	$allz(H,H[P_0+1],H_1[P+1],\langle H_1,H_1[P+1],0\rangle),P_0\neq 0$	
	$\models allz(H,P_0,H_1[P+1],\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle)$	AP 10s.1′, 10s.2a′
10s.3r.2	$allz(H,H[P_0+1],H_1[P+1],\langle H_1,H_1[P+1],0\rangle),P_0\neq 0$	
	$\models allz(H,H[P_0+1],H_1[P+1],?H_2),P_0\neq 0,$	
	$\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle=\langle ?H_2,P_0,0\rangle$	RU 10s.3r.1
10s.3r.3	$\neg\Box$	DP 10s.3r.2 $\{H_2\mapsto\langle H_1,H_1[P+1],0\rangle\}$

Figure 5.16: Proof of Subsumption in List Reset Proof

using a new variable P_0 which conceptually represents the address of the first node of the list. This is the obligation 1 of Figure 5.15.

Program 3.7 has a while loop, and so as in the proof of Sum, CUT is used for generalizing the lhs of obligation 3 into a loop invariant (in obligation 4). Here, the subsumption test for the application of CUT is the following (the residual obligation trivially holds):

$$p(2,H',P,H_f,P_f),P=P_0,H'=\langle H,P,0\rangle\models p(2,H',P,H_f,P_f),allz(H,P_0,P,H_1).$$

This subsumption test includes the recursive predicate *allz*. This obligation is easily established by right unfolding using the rule RU and the first clause of *allz*. The proof is shown in Figure

10r.1	$allz(H,P_0,Y,H_f),P_f = H_f[Y+1],P_f = 0,$ $allz(H,P_0,P,H'),P' = H'[P+1],P' \neq 0,H'' = \langle H',P',0\rangle$ $\models allz(H,P_0,?Last,H_f),P_f = H_f[?Last+1],P_f = 0$	AP 4,9
10r.2	$\neg\Box$	DP 10r.1 $\{Last \mapsto Y\}$

Figure 5.17: Proof of Residual Obligation in List Reset Proof

4.1	$p(2,H',P,H_f,P_f),P \neq 0,P = P_0,H' = \langle H,P,0\rangle$ $\models p(2,H',P,H_f,P_f),allz(H,P_0,P,H')$	CUT 4
4.1$'$	$p(2,H',P,H_f,P_f),P \neq 0$ $\models p(2,H',P,H_f,P_f),allz(H,P,P,\langle H,P,0\rangle)$	Simplified 4.1
4.2	$p(2,H',P,H_f,P_f),P \neq 0$ $\models p(2,H',P,H_f,P_f),P \neq 0$	RU 4.1$'$
4.3	$\neg\Box$	DP 4.2

Figure 5.18: Proof of CUT Condition in List Reset Proof

5.18.

Here we notice that each iteration of the loop changes the distance between the pointer P and the head P_0 of the original list. The generalization into the lhs atom $allz(H,P_0,P,H')$ in obligation 4 represents the relationship between P and P_0 in any iteration throughout the execution of the loop.

Now we continue the proof of obligation 4 of Figure 5.15. Further left unfolds will result in branching into two obligations, one represents the exit and reaching of the final program point Ω (obligation 6a), while the other represents the re-entry of the loop (obligation 6b).

From 6b we obtain obligation 9 through a left unfolding step. We prove 9 by applying 4 as an induction hypothesis via the AP rule. For AP to be applicable, one of the requirement is for us to prove the following subsumption test:

$$allz(H,P_0,P,H'),P' = H'[P+1],P' \neq 0,H'' = \langle H',P',0\rangle \models allz(H,P_0,P',H'').$$

This assertion proves the actual invariance of the loop invariant which has been used to strengthen obligation 3 into obligation 4 using CUT. The proof of this assertion is shown in Figure 5.16.

We note that we have used Program 4.3 as the definition of *allz* in Figure 5.16. Program proofs about assertions that are specified recursively usually require that the program fragment behaves in tandem with the recursive formulation. That is, the program fragment increments the data structure in the manner specified in the assertion for incrementally larger data structures.

D.1	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I), no\_share(H,J,I),$ $I \neq 0, H' = \langle H,I+1,J\rangle, I' = H[I+1], J' = I$ $\models no\_share(H',J',I')$	
D.1′	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I), no\_share(H,J,I), I \neq 0$ $\models no\_share(\langle H,I+1,J\rangle, I, H[I+1])$	Simplified D.1
D.2	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,H[I+1]), no\_reach(H,I,H[I+1]),$ $no\_share(H,J,I), I \neq 0$ $\models no\_share(\langle H,I+1,J\rangle, I, H[I+1])$	LU D.1′
D.3	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,H[I+1]), no\_reach(H,I,H[I+1]),$ $no\_share(H,J,I), I \neq 0$ $\models no\_reach(\langle H,I+1,J\rangle, I, H[I+1]),$ $no\_share(\langle H,I+1,J\rangle, \langle H,I+1,J\rangle[I+1], H[I+1]), I \neq 0$	RU D.2
D.4	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,H[I+1]), no\_reach(H,I,H[I+1]),$ $no\_share(H,J,I), I \neq 0$ $\models no\_reach(\langle H,I+1,J\rangle, I, H[I+1]),$ $no\_share(\langle H,I+1,J\rangle, J, H[I+1]), I \neq 0$	AIP D.3
D.5	$reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,H[I+1]), no\_reach(H,I,H[I+1]),$ $no\_share(H,J,I), I \neq 0$ $\models no\_reach(\langle H,I+1,J\rangle, I, H[I+1]),$ $no\_share(H,J,H[I+1]), I \neq 0$	SEP D.4
D.6	$\neg\Box$	DP D.5 with F.1

Figure 5.19: Proof of Assertion D

For example, in the proof above, the *allz* predicate could have instead been specified as Program 4.4.

Program 4.4 is "sublist-recursive" in the sense that a zeroed list segment starting from the node with address X, and ending at the node with address Y is defined to be a zeroed list segment from address X to T, appended by one extra zero node at address Y. However, in the previous section, we in fact used Program 4.3 as the definition of *allz*. Program 4.3 which we have used in the proof is "tail-recursive," that is, the zeroing of a list is specified in terms of the zeroing of its tail. The proof is actually easier if Program 4.4 is used.

The property that there is no strong dependency on how the assertion predicate is defined allows for greater flexibility. This is essentially enabled by coinduction. Notice that in Figure 5.16 we have used the AP rule to complete the proof. This is not necessary had we used the "sublist-recursive" definition of *allz*.

5.9.3 Handling Separation: List Reverse

In our framework, we can also state that two data structures are "separate," that is, there is no common cell that is reachable from both. This is done by using the *no_reach* predicates (Program

4.10 or Program 3.11) and the *no_share* predicate (Program 4.13). In this section we demonstrate the use of SEP together with *no_share* to complete a proof.

Here we use the motivating example of [166], which is on proving acyclic list reversal. Consider again Program 4.5 with its correctness statement (4.3) given in Section 4.4, as follows:

$$p(0,H,I,J,H_f,J_f), alist(H,I), J=0 \models reverse(H,I,0,H_f,J_f), alist(H_f,J_f).$$

The assertion says that given an acyclic list with head I as input, we get as output a list with head J, which is a reverse of the original list. In (4.3), J_f denotes the final value of the variable J, and H_f denotes the final state of the heap. The correctness statement here requires a reference to the input variables (H and I), which is easily specified using our assertion language.

As with the proof of list reset program presented earlier, the main proof of the list reverse program is again similar in structure to the basic while loop program Sum given in Section 5.5. We therefore relegate the complete proof to the appendix Section B.1.1.

As with Sum and list reset, the proof requires an introduction of loop invariant in order to find a recursion in the unfolding of the loop. Here we again use the CUT rule to introduce loop invariant, which requires us to prove the following assertion:

$$\begin{aligned} & p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0 \\ & \quad \models p(0,H,I,J,H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), \\ & \qquad alist(H,I), no\_share(H,J,I). \end{aligned} \tag{5.14}$$

The proof is shown in the appendix Section B.1.2.

Note that the loop invariant (5.14) states, amongst other things, the key property that the lists α (which address is I) and β (which address is J) are separate in memory by the predicate $no\_share(H,J,I)$. One iteration of the loop body produces new lists $\alpha' = tail(\alpha)$ and $\beta' = head(\alpha)\cdot\beta$ (where $\cdot$ denotes a concatenation). This modification is the result of the update of the heap from H to $H' = \langle H, I+1, J\rangle$. We want to prove that the new lists α' and β' are also separated in H'. This is expressed by the following assertion D, which is one of the assertions required to prove the side condition of the application of AP in the main proof (appendix Section B.1.1).

$$\begin{aligned} & reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I), no\_share(H,J,I), \\ & \quad I \neq 0, H' = \langle H, I+1, J\rangle, I' = H[I+1], J' = I \models no\_share(H',J',I'). \end{aligned}$$

The proof of D uses separation principle (SEP), and it requires F (proof in appendix Section B.1.8), which in turn requires G (proof in appendix Section B.1.9).

We present the proof in Figure 5.19. Here, the separation principle is used to simplify the atom $\textit{no\_share}(\langle H, I+1, J\rangle, J, H[I+1])$ in the rhs of (D.4) into the atom $\textit{no\_share}(H, J, H[I+1])$ in (D.5). This simplification can be inferred from the atoms $\textit{no\_reach}(H, I, H[I+1])$ and $\textit{no\_share}(H, J, I)$ of the lhs of (D.5).

We now explain the intuitive proof of the assertion. Notice that we have that $\textit{no\_share}(H, J, I)$ and because α is acyclic ($\textit{alist}(H, I)$) and nonempty ($I \neq 0$) we also have that $\textit{no\_reach}(H, I, H[I+1])$ (this reasoning corresponds to application of LU to (D.1$'$) producing D.2 in Figure 5.19). Hence here we know that $\textit{head}(\alpha)$ is separated from $\textit{tail}(\alpha)$ and β, and we can therefore reason that $\alpha' = \textit{tail}(\alpha)$ and $\beta' = \textit{head}(\alpha) \cdot \beta$ are separated (or, $\textit{no\_share}(H', J', I')$ holds).

5.9.4 Intermittent Abstraction Solves Intermittence Problem

We demonstrate here the use of intermittent abstraction in solving intermittence problem in data structure, and also quantitative reasoning on an abstract data structure. We use the example of Rugina [173], which has been introduced in Section 4.4 (Program 4.14 with CLP model Program 4.15).

AVL is a balanced binary tree, where for each node, the depth of its left and right subtrees differs by only one node. The recursive specification of an AVL tree is given as Program 4.16.

Again we repeat the correctness of the AVL tree from Section 4.4 as follows.

$$\begin{aligned}
&p(0, H, X, Y, Z, H_f, Y_f), \textit{avltree}(H, H[X+2], DL-2),\\
&\quad \textit{avltree}(H, H[H[X+1]+1], DL-1), \textit{avltree}(H, H[H[X+1]+2], DL-2),\\
&\quad \textit{no\_share}(H, X, H[X+2], H[H[X+1]+1], H[H[X+2]+2])\\
&\qquad \models \textit{avltree}(H_f, Y_f, DL).
\end{aligned} \tag{5.15}$$

In the above, the depth of the left subtree of the input tree is denoted by the variable DL.

Program 4.14 is given an input an unbalanced subtree rooted at x, where its left subtree is two deeper than its right subtree, and at its left child, the left subtree is 1 deeper than its right subtree. As the output, we expect to obtain a balanced AVL tree. However, right at program point $\langle 6\rangle$, the structure becomes temporarily cyclic, hence here it no longer makes sense to speak about depth of left and right subtree. In our proof method, this is not a problem due to our intermittent abstraction: We do not have to abstract the state at $\langle 6\rangle$ in the same way as shape analysis.

In the proof, we perform left unfold repeatedly from (5.15), which represents the state at program point $\langle 0 \rangle$, according to the program until we reach program point $\langle 7 \rangle$. The last obligation generated is the following.

$$
\begin{array}{l}
p(7,H_f,X,Y_f,H_2[Y_f+2],H_f,Y_f),Y_f = H[X+1], \\
H[Y_f] = 1,H_1 = \langle H,X,0\rangle,H_2 = \langle H_1,Y_f,0\rangle, \\
H_f = \langle H_2,Y_f+2,X\rangle,H_f = \langle H_f,X+1,H_2[Y_f+2]\rangle, \\
avltree(H,H[X+2],DL-2), \\
avltree(H,H[H[X+1]+1],DL-1), \\
avltree(H,H[H[X+1]+2],DL-2), \\
no\_share(H,X,H[X+2],H[H[X+1]+1],H[H[X+2]+2]) \\
\models avltree(H_f,Y_f,DL).
\end{array}
$$

From here we perform right unfold according to the recursive definition of *avltree* obtaining the assertion

$$
\begin{array}{l}
p(7,H_f,X,Y_f,H_2[Y_f+2],H_f,Y_f),Y_f = H[X+1], \\
H[Y_f] = 1,H1 = \langle H,X,0\rangle,H_2 = \langle H_1,Y_f,0\rangle, \\
H_f = \langle H_2,Y_f+2,X\rangle,H_f = \langle H_f,X+1,H_2[Y_f+2]\rangle, \\
avltree(H,H[X+2],DL-2), \\
avltree(H,H[H[X+1]+1],DL-1), \\
avltree(H,H[H[X+1]+2],DL-2), \\
no\_share(H,X,H[X+2],H[H[X+1]+1],H[H[X+2]+2]) \\
\models \\
H_f[Y_f] = 0,H_f[H_f[Y_f+2]] = 0, \\
avltree(H_f,H_f[Y_f+1],DL-1), \\
avltree(H_f,H_f[H_f[Y_f+2]+1],DL-2), \\
avltree(H_f,H_f[H_f[Y_f+2]+2],DL-2).
\end{array}
$$

Next we perform simplifications using the (SEP) and (AIP) principles. We use both (SEP) and (AIP) in reasoning about rhs atoms, and we use only (AIP) to prove rhs constraints. For example, we need, as a subproof, the proof of the following assertion:

$$
\begin{aligned}
& Y_f = H[X+1], H[Y_f] = 1, H_1 = \langle H, X, 0\rangle, H_2 = \langle H_1, Y_f, 0\rangle, \\
& H_f = \langle H_2, Y_f + 2, X\rangle, H_f = \langle H_f, X+1, H_2[Y_f + 2]\rangle, \\
& \mathit{no\_share}(H, X, H[X+2], H[H[X+1]+1], H[H[X+2]+2]) \\
& \models Hf[Hf[Y_f + 2]] = 0
\end{aligned}
$$

The proof is as follows:

$$
\begin{aligned}
& H_f[H_f[H[X+1]+2]] \\
= \; & H_f[\langle H_f, X+1, H_2[H[X+1]+2]\rangle[H[X+1]+2]] \\
= \; & H_f[H_f[H[X+1]+2]](\text{since } X+1 \neq H[X+1]+2) \\
= \; & H_f[\langle H_2, H[X+1]+2, X\rangle[H[X+1]+2]] \\
= \; & H_f[X] \\
= \; & \langle H_f, X+1, H_2[H[X+1]+2]\rangle[X] \\
= \; & H_f[X](\text{since } X+1 \neq X) \\
= \; & \langle H_2, H[X+1]+2, X\rangle[X] \\
= \; & H_2[X](\text{since } H[X+1]+2 \neq X) \\
= \; & \langle H_1, H[X+1], 0\rangle[X] \\
= \; & H_1[X](\text{since } H[X+1] \neq X) \\
= \; & \langle H, X, 0\rangle[X] = 0
\end{aligned}
$$

Note that the premise $H[X+1] \neq X$ is justified by the existence of the *no_share* predicate on the lhs.

The assertions (4.4) is also proved similarly, using (AIP) and (SEP) principles. The same techniques are also applicable for the correctness proof of bubble sort (4.1). We do not give the proofs here.

5.10 Discussion

5.10.1 Comparison to Mesnard et al.'s Proof Method

Mesnard et al. [141] propose a proof method for constraint logic programs to prove a system of implications whose consequents only contain constraints. The method is based on transformation, where we transform each clause of the given CLP program into a constraint whose

$$\begin{array}{lll} p(0). & & \kappa_1 \\ p(X) & \texttt{:-}\ \ 2X = 1. & \kappa_2 \end{array}$$

Program 5.4: Mesnard et al.'s Example I

$$\begin{array}{r|ll} 1 & p(X) \models X = 0 & \\ \hline 2a & X = 0 \models X = 0 & \textsc{lu}\ 1 \\ 2b & 2X = 1 \models X = 0 & \textsc{lu}\ 1 \\ 3 & \neg\Box & \textsc{dp}\ 2a \\ 4 & \neg\Box & \textsc{dp}\ 2b \end{array}$$

Figure 5.20: Proof of Mesnard et al.'s Example I

unsatisfiability implies that the system of implications hold. The method has a relatively weak completeness result, which we believe is due to the use of a specific induction schema which forces the application of induction in one unfold step. Mesnard et al. provide three verification examples which cannot be solved using their method. One of the example pertains to the imprecision of the chosen constraint domain, which is also inherent in our proof method, and hence we also do not solve this problem. However, the other incompleteness cases are due to the proof method itself, and here we show how we may prove them using our method.

The first example demonstrates the inherent incompleteness due to the actual constraint domain used in the implementation. Given Program 5.4, the assertion $p(X) \models X = 0$ holds in CLP($\mathbb{N}$). The proof in CLP($\mathbb{N}$) is in Figure 5.20. However, the assertion does not hold in CLP($\mathbb{Q}$). Our implementation to be described in Chapter 8 using CLP($\mathcal{R}$) system to verify integer assertions is therefore necessarily incomplete.

Similar to ours, the proof method of [141] is also inductive. However, we can say that in their proof method, induction hypothesis has to be applied after just one level of unfold (corresponding to the application LU). Here given Program 5.5, we would like to prove the following system of implications:

$$\begin{array}{rcl} q(X) & \models & X = X \\ p(X,Y) & \models & X = fgh(X), Y = hfg(Y). \end{array}$$

The first assertion trivially holds, and only provided in [141] as a comparison with the use of the stronger assertion $q(X) \models X = ghf(X)$.

The proof of the second assertion requires two level of unfolds for induction hypothesis to be applicable, and therefore cannot be done using Mesnard et al.'s proof method. Our proof of the

$$\begin{array}{lll}
q(U) \ \texttt{:-}\ U = g(V), V = h(W), W = f(U). & \kappa_1 \\
q(U) \ \texttt{:-}\ U = g(V), p(f(U), V). & \kappa_2 \\
p(U,V) \ \texttt{:-}\ V = h(U), q(g(V)). & \kappa_3
\end{array}$$

Program 5.5: Mesnard et al.'s Example II

$$\begin{array}{r|ll}
1 & p(X,Y) \models X = fgh(X), Y = hfg(Y) & \\ \hline
2 & Y = h(X), q(g(Y)) \models X = fgh(X), Y = hfg(Y) & \textsc{lu}\ 1 \\
3a & Y = h(X), g(Y) = ghfg(Y) \models X = fgh(X), Y = hfg(Y) & \textsc{lu}\ 2 \\
3b & Y = h(X), g(Y) = g(V), p(fg(Y), V) \models X = fgh(X), Y = hfg(Y) & \textsc{lu}\ 2 \\
4 & \neg\Box & \textsc{dp}\ 3a \\
5 & \neg\Box & \textsc{ap}\ 1{,}3b
\end{array}$$

$$\begin{array}{r|ll}
5s.1 & Y = h(X), g(Y) = g(V), p(fg(Y), V) \models p(fgh(X), V) & \textsc{ap}\ 1{,}3b \\ \hline
5s.2 & \neg\Box & \textsc{dp}\ 5s.1
\end{array}$$

$$\begin{array}{r|ll}
5r.1 & Y = h(X), g(Y) = g(V), fg(Y) = fghfg(Y), V = hfg(V) & \\
 & \qquad \models X = fgh(X), Y = hfg(Y) & \textsc{ap}\ 1{,}3b \\ \hline
5r.2 & \neg\Box & \textsc{dp}\ 5r.1
\end{array}$$

Figure 5.21: Proof of Mesnard et al.'s Example II

assertion is shown in Figure 5.21.

Mesnard et al.'s proof method requires a demonstration of inductive proof from constraints only. For example, suppose that we want to prove the assertion $p(X,Y) \models Y+2 \leq 3X$ holds on Program 5.6. In Mesnard et al.'s proof method, this would be transformed into the unsatisfiability questions of the following two goals:

$$\begin{array}{l}
X = 1, Y = 1, \neg(Y+2 \leq 3X) \\
X = X'+1, Y = Y'+2X'+1, Y'+2 \leq 3X', \neg(Y+2 \leq 3X)
\end{array} \tag{5.16}$$

The second goal is satisfiable in the integer domain, hence the proof method (luckily) correctly concludes that the initial assertion $p(X,Y) \models Y+2 \leq 3X$ does not hold. However, this conclusion may be wrong since instead the second goal above, the actual proof that we need is the unsatisfiability of the goal

$$p(X',Y'), X = X'+1, Y = Y'+2X'+1, Y'+2 \leq 3X', \neg(Y+2 \leq 3X),$$

that is, with $p(X',Y')$ included.

The problem here is that Mesnard et al.'s transformation is similar to an application of LU

$$\begin{array}{lll} p(1,1). & & \kappa_1 \\ p(X+1, Y+2X+1) & \texttt{:-} \; p(X,Y). & \kappa_2 \end{array}$$

Program 5.6: Mesnard et al.'s Example III

1	$p(X,Y) \models Y+2 \leq 3X$	
2a	$X=1, Y=1 \models Y+2 \leq 3X$	LU 1
2b	$p(X',Y'), X=X'+1, Y=Y'+2X'+1 \models Y+2 \leq 3X$	LU 1
3	$\neg\Box$	DP 2a
4	$\neg\Box$	AP 1,2b

4s.1	$p(X',Y'), X=X'+1, Y=Y'+2X'+1 \models p(X',Y')$	AP 1,2b
4s.2	$\neg\Box$	DP 4s.1

Figure 5.22: Partial Refutation of Mesnard et al.'s Example III

once followed by AP. The application of AP spawns two new obligations: the subsumption test and the residual obligation. These two obligations are only sufficient conditions. If either one of them does not hold, we cannot conclude that the initial obligation does not hold.

Let us explain this more carefully using our proof method, by examining the partial refutation in Figure 5.22. We have not finished the proof, but here some explanations are necessary. Essentially from the initial obligation 1 we obtain two new proof obligations: 2a and 2b, each corresponds to Mesnard et al.'s goals (5.16). The first obligation is exactly the negation of Mesnard et al.'s first goal, and it can be proved immediately (since Mesnard et al.'s first goal is unsatisfiable). Now, we apply AP on 2b, resulting in the subsumption 4s.1. The residual obligation (4r.1) here is

$$Y'+2 \leq 3X', X=X'+1, Y=Y'+2X'+1 \models Y+2 \leq 3X.$$

Notice that this is exactly the negation of Mesnard et al.'s second goal.

Now, here, although 4r.1 does not hold (or, Mesnard et al.'s second obligation is satisfiable), we cannot really conclude that the original assertion $p(X,Y) \models Y+2 \leq 3X$ does not hold.

The point of Mesnard et al.'s argument in [141] is that had we included the predicate $p(X',Y')$ in the lhs of the obligation 4r.1, and the obligation is false (possibly proved by further unfolds), then we can conclude that the target obligation is false. This is because the inclusion of $p(X',Y')$ in the lhs would make 4r.1 and 4s.1 no longer sufficient, but exact conditions.

Using our proof method, however, we can find a true refutation of the target obligation using

1	$p(X,Y) \models Y+2 \leq 3X$	
2a	$X=1, Y=1 \models Y+2 \leq 3X$	LU 1
2b	$p(X',Y'), X=X'+1, Y=Y'+2X'+1 \models Y+2 \leq 3X$	LU 1
3	$\neg\Box$	DP 2a
4a	$X'=1, Y'=1, X=X'+1, Y=Y'+2X'+1 \models Y+2 \leq 3X$	LU 2b
4b	$p(X'',Y''), X'=X''+1, Y'=Y''+2X''+1, X=X'+1, Y=Y'+2X'+1$	
	$\models Y+2 \leq 3X$	LU 2b
5	$\neg\Box$	DP 4a
6a	$X''=1, Y''=1,$	
	$X'=X''+1, Y'=Y''+2X''+1, X=X'+1, Y=Y'+2X'+1$	
	$\models Y+2 \leq 3X$	LU 4b
6b	$p(X''',Y'''), X''=X'''+1, Y''=Y'''+2X'''+1,$	
	$X'=X''+1, Y'=Y''+2X''+1, X=X'+1, Y=Y'+2X'+1$	
	$\models Y+2 \leq 3X$	LU 4b

Figure 5.23: Full Refutation of Mesnard et al.'s Example III

only the DP and LU rules. This is shown by the refutation in Figure 5.23, where the obligation 4a is false.

5.10.2 On Manna-Pnueli's Universal Invariance Rule

Here we show how our coinductive proof method is related to Manna-Pnueli's universal invariance rule [136], a well-known inductive proof technique for programs, which is an instance of computational induction.

In our proof method, we generally start from an initial assertion $p(\tilde{X}), q_l(\tilde{X}) \models q_r(\tilde{X})$, but this is just logically equivalent to $p(\tilde{X}) \models (q_l(\tilde{X}) \Rightarrow q_r(\tilde{X}))$, and hence we can replace the rhs with just some predicate $q(\tilde{X})$, which holds if and only if $q_l(\tilde{X}) \Rightarrow q_r(\tilde{X})$ and hence, in logical sense, we can always assume that what we want to prove is an assertion $p(\tilde{X}) \models q(\tilde{X})$. Here, assuming a backward CLP model of programs, $p(\tilde{X})$ represents any state of the program. Our assertion therefore states $\varphi(\tilde{X})$ is satisfied in any state $p(\tilde{X})$ of the program. This is known as an *invariance property*.

A well-known technique for proving invariance is due to Manna and Pnueli [136]. The rule is called the *universal invariance rule* (*A-INV rule*). The formulation below is following [160].

Other versions are presented in [136, 19]:

$$\frac{\begin{array}{ll} \text{I1.} & \Theta \Rightarrow \varphi \\ \text{I2.} & \varphi \Rightarrow q \\ \text{I3.} & \varphi \wedge \rho \Rightarrow \varphi' \end{array}}{\Theta \models \Box q}$$

For readers unfamiliar with temporal logic, the CTL conclusion $\Theta \models \Box q$ means that q holds in all reachable states of the program, when execution starts from the initial state satisfying Θ. The formula φ is called an *inductive invariant*. Similar inductive proof rule without inductive invariant is given by Misra in [144].

Here we demonstrate how we may perform inductive proof using universal invariance rule above in our framework. We show how our proof rules derive the premises I1, I2, and I3 of the A-INV rule from the original verification question.

We first assume that we have the following program, which is a backward model of a transition system in CLP with an initial state and n transitions.

$$\begin{array}{lll} p(\tilde{X}) & :- & \Theta(\tilde{X}). \\ p(\tilde{X}') & :- & \rho_1(\tilde{X},\tilde{X}'), p(\tilde{X}). \\ & \vdots & \\ p(\tilde{X}') & :- & \rho_{n-1}(\tilde{X},\tilde{X}'), p(\tilde{X}). \end{array}$$

Again, here we prove the invariance property $p(\tilde{X}) \models q(\tilde{X})$.

Using our CUT rule, we may introduce or strengthen an inductive invariant by strengthening an assertion. Here, we replace the assertion $p(\tilde{X}) \models q(\tilde{X})$ with the assertion $\varphi(\tilde{X}) \models q(\tilde{X})$. The latter assertion corresponds to Premise I2. For this we are required to prove $(\varphi(\tilde{X}) \models q(\tilde{X})) \rhd (p(\tilde{X}) \models q(\tilde{X}))$, which again consists of the proofs of the following two:

- Subsumption: $p(\tilde{X}) \models \varphi(\tilde{X})$.
- Residual assertion: $q(\tilde{X}) \models q(\tilde{X})$.

The residual assertion obviously holds, and can immediately be discharged.

We apply LU to the subsumption test above. The unfold using the CLP fact $p(\tilde{X})\ :-\ \Theta(\tilde{X})$. results in the obligation $\Theta(\tilde{X}) \models \varphi(\tilde{X})$ which corresponds to the Premise I1.

We will now discuss how the unfold followed by the application of AP, using all of the other CLP clauses results in obligations which correspond to the Premise I3.

First we apply left unfolding (LU) to the above subsumption test using the second to the last clause of the CLP program resulting in the obligations

$$\begin{array}{c} p(\tilde{X}'), \rho_1(\tilde{X}', \tilde{X}) \models \varphi(\tilde{X}) \\ \vdots \\ p(\tilde{X}'), \rho_{n-1}(\tilde{X}', \tilde{X}) \models \varphi(\tilde{X}) \end{array}$$

We now prove each of these obligations using AP. That is, for each i where $1 \leq i \leq n-1$, we prove $(p(\tilde{X}) \models \varphi(\tilde{X})) \rhd (p(\tilde{X}'), \rho_i(\tilde{X}', \tilde{X}) \models \varphi(\tilde{X}))$. Here also we prove both of the following obligations:

- Subsumption: $p(\tilde{X}'), \rho_i(\tilde{X}', \tilde{X}) \models p(\tilde{X}')$.
- Residual assertion: $\varphi(\tilde{X}'), \rho_i(\tilde{X}', \tilde{X}) \models \varphi(\tilde{X})$.

It is easy to see that the subsumption part holds. The proof of all the residual obligations for all i such that $1 \leq i \leq n-1$ constitute the proof of the Premise I3.

In forward CLP model, the constraint facts correspond to the "point of interest" of a program. Moreover, with our forward model, in proving invariance of transitions we show that assuming the postcondition satisfies φ, the precondition also satisfies φ. It is harder to think of invariants in this way, but actually here we prove that φ is invariant in all states where the "point of interest" is reachable. That is, we are actually establishing the past linear time temporal logic property $p \Rightarrow \boxminus q$ or its branching time version $p \Rightarrow \boxminus q$, where p represents the point of interest, and $\boxminus$ is the *always in the past* operator (see [19]).

5.10.3 Proving General Equivalence

In Section 4.6.2 we have shown how we may specify an equivalence property pertaining to a simple program. In this section we discuss how we may prove the equivalence using our proof method. We repeat the CLP program (Program 4.28) and the assertions to be established in Figure 5.24. The proof itself is shown in Figure 5.25 using scope notation. This example demonstrates that it is straightforward to use a proof method for implication (as is ours) in order to prove equivalence. Direct equivalence proof may be more compact than ours, but not necessarily easier.

$$\begin{array}{lll} s(\omega,\omega). & & \textbf{Assertions:} \\ s(X,X_f) & \text{:-}\ X \neq \omega, p(X)=1, X_f = X. & s(X,Y), s(Y,X_f) \models s(X,X_f) \\ s(X,X_f) & \text{:-}\ X \neq \omega, p(X)=0, s(h(X),Y), s(Y,X_f). & s(X,X_f) \models s(X,?Y), s(?Y,X_f) \end{array}$$

Figure 5.24: Example 12 of [135] and Idempotence Property

5.11 Related Work

Our proof method is closely related to various verification methods for (constraint) logic programs. Recall that we have discussed the approach of Mesnard et al. in Section 5.7.2. Here we will discuss other approaches, but before proceeding in more detail, we first summarize the following two basic advantages over any other existing proof methods:

1. Some inductive proof methods are based on fitting in the allowable inductive proofs into an *induction schema*, which is usually syntax-based. Instead, we employ no induction schema. We detect the point where we apply the induction hypothesis using subsumption. In other words, we discover the induction schema dynamically using indefinite steps of complete left unfolds. This approach is more complete and automatable.

2. We provide a generalization step (the CUT rule) which adds into the completeness of our proof method.

Most related to our proof method are the works of Kanamori and Fujita [118], and Kanamori and Seki [119]. Our rhs unfold corresponds to the *definite clause inference* (*DCI*) step, while our complete lhs unfold corresponds to *negation as failure inference* (*NFI*) of [119]. However, the main difference is in the application of induction. Here the applicability of induction, however, is limited by the lack of a generalization step (allowed by our CUT rule) and the necessity of its application in a single unfold step. A use of a kind of structural induction in a similar framework to Fujita and Seki's is demonstrated by Fribourg in [74].

Stickel proposes a Prolog-based theorem prover that is complete for for first-order predicate calculus, called *Prolog Technology Theorem Prover* (*PTTP*) [187]. The proof process is basically Prolog's refutation, that is, finding a counterexample to a query. Stickel proposes several extensions to Prolog for this purpose, including a *model elimination reduction* (*ME reduction*). Here, when reduction is applied to a literal, the original literal is stored. Whenever a new goal which is contradictory to a stored literal is found, we stop because this constitutes a refutation. Stickel's approach is similar to ours when we prove the assertion $p_1(X),\ldots,p_n(X) \models \Box$. The part of PTTP that is akin to our induction is the detection when there is an occurrence of the same literal (a kind of subsumption test), in which case, the system backtracks. Our proof method, however, does not deal with negative literals because transition systems modeled in CLP do not normally have negative literals.

Hsiang and Srivas propose an inductive proof method for Prolog programs [102, 103]. The main feature of the proof method is a semi-automatic generation of induction schema (in the sense, this objective is similar to those of Kanamori and Fujita [118]). The assertion to be proved is encoded in a predicate `prop`. The generation of inductive assertion is done by generating the reduct of the goals (unfolding). The termination of the unfolding is implemented by a marking mechanism on the variables. Whenever an input variables is instantiated during an unfold (in other words, we need to make a decision about its value), it is marked. In a sense, this is similar to the use of *bomblist* in the Boyer-Moore prover [22]. As is the case with Boyer-Moore prover, the induction is structural. However, the method lacks a generalization step. Moreover, it requires the user to distinguish a set of *input* variables to structurally induct on.

The work of Craciunescu [36] is on proving the equivalence of CLP programs using either induction or coinduction. The notion of coinduction here is different from ours. While our coinduction is a *least* fixpoint induction, the coinduction of Craciunescu is a *greatest* fixpoint induction. Greatest fixpoint induction can be used for reasoning about possible infinite computations that have no start [143]. For each induction and coinduction, Craciunescu presents separate sets of proof rules (although most rules are shared). He also proves that each method is as powerful as another. In his proof framework, a CLP program is first transformed into a CLP∀ program, which is its Clark completion (Section 2.7). The proof rules include LU and RU-like rules of the CLP∀ program. The inductive proof of Craciunescu is similar to the inductive proof of our proof method, although both have been developed independently. However, in contrast to ours, Craciunescu does not report any completed mechanization.

Also related are verification methods which are based on unfold/fold logic program transformation (of Tamaki and Sato, see [170] for an outline), notably the work of Pettorossi and Proietti in proving equivalence [157], and the work of Roychoudhury et al. which is a proof method for equivalence assertions on parameterized systems represented as logic programs [171, 172, 170]. For this purpose, Roychoudhury et al. develop a more general notion of unfold/fold, which is implemented as the SCOUT system.

Equivalence is useful to prove liveness properties, and it can be handled by our proof method by proving both ways of the implication. The SCOUT system can also be extended to prove implication [172], but the machinery for proving equivalence may not be suitable for this task because of different correctness criteria. Equivalence proof requires *total correctness*, where the transformed program has the same least model as the original.

We may also compare our proof method with unfold/fold transformation systems, by considering it as a transformation system which transforms an assertion (viewed as a Horn clause[3]), into a set of others. Now, the correctness criteria for our "transformation" system is that the resulting assertions, if they are consistent with the program, would imply the consistency of the original assertion. In a sense, the resulting assertions are "stronger" than the original. This has the implication that they have a "least model" that *subsumes* the original. This weaker correctness allows for arbitrary generalization (widening) step, as is made possible by our CUT rule, such as the intermittent abstraction discussed in Section 5.8.2.

We note that the aforementioned weaker correctness criterion is not the same as the notion of *partial correctness* of unfold/fold transformation, where a transformed program has a least model which is *subsumed by* the original program. Therefore, although the CUT rule has some resemblances in its mechanical aspects with *goal replacement* in unfold/fold transformation [170], its purpose is to strengthen a "clause" (assertion) instead of replacing it with an equivalent or weaker one as with the goal replacement technique.

The work of Pettorossi et al. [158] is on proof method for closed first-order formula given a the *perfect model semantics* (see the survey [4]) of a stratified CLP program. The method is based on unfold/fold transformations. Compared to previous works such as [171], it is more general in that it handles first-order formula instead of specific form of equivalence or implication. Compared to techniques employed in theorem proving, the authors argue that the idea provide a way to eliminate existential quantification through program transformations. The elimination here is of variables appearing only in clause bodies.

Since this proof method allows for the proof of stratified program, it is more general than ours since we handle only positive programs (CLP programs without negation). In our proof method, we do not provide any method for eliminating existential quantification of variables in the premise of an assertion. This is because such quantification is actually a universal one, since

[3]This is possible assuming the rhs of an assertion is a conjunction and does not contain existentially-quantified variables. An existential quantification is essentially a (possibly infinite) disjunction, and a disjunctive rhs would give us a non-Horn clause.

the premise is the negated part of the assertion. To eliminate the existential quantification of the conclusion, we use substitution in the DP rule. The method of Pettorossi et al. also has the limitation to real constraints due to unfold/fold technique used to eliminate existential quantifiers. In contrast, we our proof method does not have the corresponding limitation.

As an induction-based technique, our proof method is related to *fixpoint induction* [152] for proving properties of the least fixpoint of monotonic functions, such as recursive programs. A complementary technique which is also discussed in [152] can be used to reason about greatest fixpoint, which is related to the proof method of Gupta et al. [180] and the greatest fixpoint induction of Craciunescu mentioned above.

We finally compare our proof method with the well-known Boyer-Moore prover [22, 23] for functional programs. To detect the applicability of induction hypothesis, the Boyer-Moore prover uses a heuristic [24]. The technique basically detects an argument of an unfolded atom becoming specialized, denoting a decreasing measure [22, 23]. Our constraint subsumption is different, in which the detection is on all arguments of the unfolded atoms, instead of just one of the argument. This solves the `REVERSE1` problem in [22], where the Boyer-Moore prover fails to detect the applicability of induction hypothesis.

1	$s(X,Y), s(Y,X_f) \models s(X,X_f)$	
2	$s(X,X_f) \models s(X,?Y), s(?Y,X_f)$	
3a	$s(\omega,X_f) \models s(\omega,X_f)$	LU 1
3b	$X \neq \omega, p(X) = 1, X = Y, s(Y,X_f) \models s(X,X_f)$	LU 1
3c	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,Y), s(Y,X_f) \models s(X,X_f)$	LU 1
	$\neg\Box$	DP 3a
	$\neg\Box$	DP 3b
4	$\neg\Box$	AP 1,3c
5a	$X = X_f = \omega \models s(X,?Y), s(?Y,X_f)$	LU 2
5b	$X \neq \omega, p(X) = 1, X_f = X \models s(X,?Y), s(?Y,X_f)$	LU 2
5c	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,X_f) \models s(X,?Y), s(?Y,X_f)$	LU 2
6	$X = X_f = \omega \models X = ?Y = \omega, s(?Y,X_f)$	RU 5a
7	$X = X_f = \omega \models X = ?Y = \omega, ?Y = X_f = \omega$	RU 6
8	$\neg\Box$	DP 7
9	$X \neq \omega, p(X) = 1, X_f = X \models X \neq \omega, p(X) = 1, X = ?Y, s(?Y,X_f)$	RU 5b
10	$X \neq \omega, p(X) = 1, X_f = X$ $\models X \neq \omega, p(X) = 1, X = ?Y, ?Y \neq \omega, p(?Y) = 1, X_f = ?Y$	RU 9
11	$\neg\Box$	DP 10
12	$\neg\Box$	AP 2,5c

4s.1	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,Y), s(Y,X_f)$ $\models s(Z,Y), s(Y,X_f)$	AP 1,3c
4s.2	$\neg\Box$	DP 4s.1

4r.1	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,X_f) \models s(X,X_f)$	AP 1,3c
4r.2	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,X_f)$ $\models X \neq \omega, p(X) = 0, s(h(X),?Y), s(?Y,X_f)$	RU 4r.1
4r.3	$\neg\Box$	DP 4r.2 $\{Y \mapsto Z\}$

12s.1	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,X_f) \models s(Z,X_f)$	AP 2,5c
12s.2	$\neg\Box$	DP 12s.1

12r.1	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,W), s(W,X_f)$ $\models s(X,?Y), s(?Y,X_f)$	AP 2,5c
12r.2	$X \neq \omega, p(X) = 0, s(h(X),Z), s(Z,W), s(W,X_f)$ $\models X \neq \omega, p(X) = 0, s(h(X),?U), s(?U,?Y), s(?Y,X_f)$	RU 12r.1
12r.3	$\neg\Box$	DP 12r.2 $\{U \mapsto Z, Y \mapsto W\}$

Figure 5.25: Proof of Example 12 of [135]

Chapter 6

Basic Algorithm for Non-Recursive Assertions Based on Dynamic Summarization

In this chapter we propose an algorithm based on our proof method, which accommodates program verification and analysis. The main component of this algorithm is an efficient exact symbolic propagation using *dynamic summarization*. The dynamic summarization technique reformulated as computation of *Craig interpolants* [37] has been presented in [112] in the context of dynamic programming search.

This chapter is structured as follows. In Section 6.1 we first present a number of simple algorithms based on our proof rules. We then introduce the concept of dynamic summarization in Section 6.2 to make exact propagation more efficient. The dynamic summarization technique can also be used to discover a safety property of a program. We finally present our general algorithm in Section 6.3.

We note that in this chapter we mainly deal with the proof of non-recursive assertions. They are of the form $p(\ldots), \phi \models \psi$, where ϕ and ψ are constraints. We will deal more specifically with the automated proof of recursive assertions in Chapter 7.

6.1 Simple Algorithms for Program Verification and Analysis

In this section we provide some simple algorithms to ease ourselves to the development of our proposed main algorithm in Section 6.3. The implementation of some of the algorithms men-

```
          program
             $prove(\emptyset, G \models H)$
          end program

          proc $prove(\tilde{A}, G \models H)$
⟨1⟩          if $(G \models H$ is provable, or
                    there is $A \in \tilde{A}$
⟨2⟩                such that $A \rhd (G \models H))$ then
                return $Success$
             end if
             $\tilde{A} := \tilde{A} \cup \{G \models H\}$
             $F := unfold(G)$
             if $(F \neq \emptyset)$ then
                for each $(g \in F)$ do
                   $prove(\tilde{A}, g \models H)$
                end for
                return $Success$
             end if
             abort
          end proc
```

Figure 6.1: Straightforward Algorithm

tioned here will be discussed in Chapter 8.

A straightforward implementation of the proof rules of Section 5.4 is shown in Figure 6.1. When proving $G \models H$, the objective is to compute a (complete) unfold tree from G whose frontier, say $G_1, \ldots, G_n$, is such that for each $1 \leq i \leq n$, we can prove $G_i \models H$ directly, or via coinduction (AP). A canonical algorithm for left unfolding (LU) is thus obtained by performing unfolding step by step, and at each step, checking the new frontier goals against H and for coinduction, terminating when there are no more unresolved frontier goals. The order in which unfolding is performed, i.e. the choice of which frontier goal to unfold next can be arbitrary. We use a depth-first strategy in Figure 6.1.

The proof of $G \models H$ in Line ⟨1⟩ and the proof of $A \rhd (G \models H)$ in Line ⟨2⟩ of Figure 6.1 can only be done directly (using DP and constraint solving), since in this chapter we deal only with assertions $p(\ldots), \phi \models \psi$. In Chapter 7 we will consider more general assertions.

As has been discussed in Section 5.6, we sometimes encounter an obligation which has already been proved in some other part of the proof tree. In this case, we may immediately establish the redundant obligation. This generalizes parent-child entailment and is best supported via a global tabling mechanism. Here whenever we encounter an obligation that is redundant to one already stored in the table, we stop (return *Success*). Otherwise, we add it into the table and

```
          program
             $Table := \emptyset$
             $prove(G \models H)$
          end program

          proc $prove(G \models H)$
⟨1⟩          if ($G \models H$ is provable, or
                  there is $A \in Table$
⟨2⟩              such that $A \rhd (G \models H)$) then
                return Success
             end if
             $Table := Table \cup \{G \models H\}$
             $F := unfold(G)$
             if ($F \neq \emptyset$) then
                for each ($g \in F$) do
                   $prove(g \models H)$
                end for
                return Success
             end if
             abort
          end proc
```

Figure 6.2: Algorithm with Global Tabling

unfold (apply LU) further. We display the pseudocode of this algorithm in Figure 6.2.

So far we have not discussed how the CUT rule is implemented. Recall that the CUT rule is used in the intermittent abstraction proof method (Section 5.8.2), the program verification method (Section 5.8.3) and also in reduction by the use of relative safety (e.g., symmetry) assertions (Section 5.6). The use of CUT rule in these cases differ. In the case of intermittent abstraction and program verification, whenever we can replace the assertion $G \models H$ with the assertion $G' \models H$ where $G \models G'$ holds, we never again consider proving $G \models H$. On the other hand, in the case of symmetry reduction we only make an attempt at replacing $G \models H$ with $G' \models H$. In case $G' \models H$ cannot be immediately concluded (via AP), we revert back to proving $G \models H$. These two different cases induce two different algorithms for implementing CUT.

For the relative safety case, before we test whether $G \models H$ holds in the function *prove*, we first try to apply a set of independently proved assertions $G \models G'$ (which always include $G \models G$) to G, which we use to generalize G to G'', and then try to prove $G'' \models H$ instead by DP or AP, failing which, we continue to unfold $G \models H$. The resulting algorithm is shown in Figure 6.3.

We show our algorithm for the case of intermittent abstraction and program verification in Figure 6.4. Compared to 6.3, it includes a small addition of an if conditional at ⟨1⟩. The additional

```
        program
            $Table := \emptyset$
            $prove(G \models H)$
        end program

        proc $prove(G \models H)$
$\langle 1 \rangle$    for each $(G'' \in \{G' | G \models G'\})$ do
$\langle 2 \rangle$        if $(G'' \models H$ is provable, or
                    there is $A \in Table$
$\langle 3 \rangle$                such that $A \rhd (G'' \models H))$ then
                    return $Success$
                end if
            end for
            $Table := Table \cup \{G \models H\}$
            $F := unfold(G)$
            if $(F \neq \emptyset)$ then
                for each $(g \in F)$ do
                    $prove(g \models H)$
                end for
                return $Success$
            end if
            abort
        end proc
```

Figure 6.3: First Algorithm Using CUT and Global Tabling

if conditional is located after the test for direct proof of $G \models H$, but before the table checking. This is because when $G \models G'$, the direct proof of $G \models H$ is often easier than $G' \models H$ (e.g., when $G \equiv \Box$). The predicate *abstraction_point*(G) tests whether G matches some criteria where we can apply CUT. If *abstraction_point*(G) holds, and there is G' in an assertion $G \models G'$ supplied, and already proved independently by the user, then we continue with proving $G' \models H$ instead of $G \models H$ by assigning G' to G in $\langle 2 \rangle$.

6.2 Dynamic Summarization

In this section we discuss an optimization technique to verify programs using depth-first search strategy. This technique can be extended into one for extracting bounds (e.g., time, energy consumption, etc.) from a program. It is mainly intended as an optimization of the algorithm in Figure 6.4. We note that in Figure 6.4, the algorithm performs exact symbolic traversal between abstraction points where the unfoldings (LU applications) are not interleaved with CUT. Here we optimize the symbolic propagation. This optimization will then become a primary component of

```
        program
          $Table := \emptyset$
          $prove(G \models H)$
        end program

        proc $prove(G \models H)$
          if $(G \models H$ is provable$)$ then
            return $Success$
          end if
⟨1⟩       if $(abstraction\_point(G)$ and $G \models G')$ then
⟨2⟩         $G := G'$
          end if
          if (There is $A \in Table$
                such that $A \rhd (G \models H))$ then
            return $Success$
          end if
          $Table := Table \cup \{G \models H\}$
          $F := unfold(G)$
          if $(F \neq \emptyset)$ then
            for each $(g \in F)$ do
              $prove(g \models H)$
            end for
            return $Success$
          end if
          abort
        end proc
```

Figure 6.4: Second Algorithm Using CUT and Global Tabling

our main algorithm in Section 6.3.

The idea is based on strengthening an assertion $G \models H$, proved via a proof tree T into a stronger assertion $G' \models H'$ which can also be proved using the same proof tree T. The stronger assertion $G' \models H'$ has more chance of making other obligations redundant (by redundancy discussed in Section 5.6) than the original $G \models H$.

Before proceeding to the main discussion, we note that in this section we will discuss proof trees explicitly, and when referring to a proof tree we would employ the terms parent, child, sibling, ancestor, and descendants, which are defined as usual.

```
⟨0⟩  if (a = 1) then
⟨1⟩     skip
     end if
⟨2⟩  if (b = 1) then
⟨3⟩     c := 0
     end if
⟨4⟩  if (c = 1) then
⟨5⟩     x := x+1
     end if
```

Program 6.1: Simple If Sequence Program

6.2.1 First Example

We start with an example Program 6.1 and its CLP model Program 6.2. These programs are already presented in Section 5.8.4, but repeated here. One possible incomplete proof of

$$p(0,X,A,B,C,X_f),X > 0 \models X_f > 0$$

using depth-first strategy is shown as a tree in Figure 6.5, where assertion number indicates the order in which the assertions are produced by applications of LU or DP. An arrow augmented with with LU or DP indicates that the target assertion is obtained through an application of LU or DP respectively from the source assertion. Note that a complete proof would have assertion 2b expanded.

In Figure 6.5, whenever an assertion $G \models H$ is established via a left unfolding using some clause κ which is then followed by DP, the question is what is a stronger $\hat{G} \models \check{H}$ that we can use as a replacement such that the same left unfold using clause κ followed by DP still proves $\hat{G} \models \check{H}$. Typically, $\hat{G}$ is a generalization of G and $\check{H}$ implies, or simply H. Now, this generalization of G into $\hat{G}$ necessitates the generalization of ancestor goals as well, since only more general ancestor goals can be left-unfolded to $\hat{G}$. More concretely, when $G \models H$ is derived from $G' \models H$ where $G \in \mathit{unfold}(G')$, the generalization $\hat{G}$ would induce a generalization of $\hat{G}'$ of G' such that $\hat{G} \in \mathit{unfold}(\hat{G}')$.

Recall that our redundancy check (Section 5.6) is based on establishing subsumption. The purpose of generalization is to obtain more chance for subsumption.

Here we exemplify *constraint deletion* as an algorithm that we use for generalizing goals. At each DP proof, we delete constraints that are not necessary to establish the proof. For example, at assertion 6a in Figure 6.5, we can delete all constraints but the underlined $C' = 0$ and $C' = 1$.

Other constraints are not necessary to establish the proof by DP which produces 7. This has the consequence that at assertion 5, only $C' = 0$ is required to ensure that the proof at 6a succeeds. Similarly, at assertion 8 we can delete all constraints but $X > 0$ and $X = X_f$. This means that at 6b only $X > 0$ is important. Further, at 5 $X > 0$ is required to ensure that the proof at 8 succeeds. The set of important constraints at 5 is now consists of $X > 0$ and $C' = 0$. In this manner, we can actually strengthen assertion 5 into $p(4,X,A,B,C',X_f),X > 0,C' = 0 \models X_f > 0$ by deleting all constraints in its lhs goal except $X > 0$ and $C' = 0$.

Now, using information obtained form its children, we can strengthen assertion 3 into

$$p(2,X,A,B,C,X_f),X > 0 \models X_f > 0.$$

Assertion 3 is now stronger than 2b, that is,

$$(p(2,X,A,B,C,X_f),X > 0 \models X_f > 0) \rhd (p(2,X,A,B,C,X_f),X > 0,A \neq 1 \models X_f > 0).$$

Therefore, 2b is now redundant. In this way we obtain more redundancy than normally possible, hence reducing the proof size.

Note however that this method is opportunistic because reduction may not be applicable even after applying constraint deletions. For example, in Figure 6.5, we notice that 4b is not redundant to 5 in the way 2b is redundant to 3. Hence, we need to expand the proof subtree of 4b.

We note that our technique for goal generalization here preserves the original proof tree. For comparison, abstractions such as one introduced by our intermittent abstraction technique (Section 5.8.2) in general introduces new proof paths (called *spurious paths*), that are nonexistent without generalization. Also, this technique is applicable only to depth-first backtracking proof algorithms.

6.2.2 Summarization

Recall the definition of assertion entailment (Definition 5.2) in Chapter 5. The concept of assertion entailment is required in a proof of some assertion $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$, where we want its proof tree to also generate a stronger assertion $G \models H$ as a replacement, which we then record in the global table as proved. The stronger assertion has more chance to establish as redundant, other assertion in the proof tree. As shown in Figure 6.5, the assertion 2b is not redundant to the original obligation 3 which is $p(2,X,A,B,C,X_f),X > 0,A = 1 \models X_f > 0$, but 2b becomes

```
p(0,X,A,B,C,X_f)  :-  p(1,X,A,B,C,X_f),A = 1.
p(0,X,A,B,C,X_f)  :-  p(2,X,A,B,C,X_f),A ≠ 1.
p(1,X,A,B,C,X_f)  :-  p(2,X,A,B,C,X_f).
p(2,X,A,B,C,X_f)  :-  p(3,X,A,B,C,X_f),B = 1.
p(2,X,A,B,C,X_f)  :-  p(4,X,A,B,C,X_f),B ≠ 1.
p(3,X,A,B,C,X_f)  :-  p(4,X,A,B,0,X_f).
p(4,X,A,B,C,X_f)  :-  p(5,X,A,B,C,X_f),C = 1.
p(4,X,A,B,C,X_f)  :-  p(Ω,X,A,B,C,X_f),C ≠ 1.
p(5,X,A,B,C,X_f)  :-  p(Ω,X+1,A,B,C,X_f).
p(Ω,X,A,B,C,X).
```

Program 6.2: Simple If Sequence Program CLP Model

redundant after 3 is strengthened to $p(2,X,A,B,C,X_f),X > 0 \models X_f > 0$.

We are now ready to provide a definition for *summarization*.

Definiton 6.1 (Summarization). Given an assertion A that is proved by a proof tree T, another assertion S is a *summarization* of A with proof tree T when S can be proved to hold by the same proof tree T, and $S \rhd A$.

As in the example above, the summarization of 3 which is $p(2,X,A,B,C,X_f),X > 0,A = 1 \models X_f > 0$ is $p(2,X,A,B,C,X_f),X > 0 \models X_f > 0$. The summarization can still be proved by the proof tree of 3 and is stronger than 3, i.e., $(p(2,X,A,B,C,X_f),X > 0 \models X_f > 0) \rhd (p(2,X,A,B,C,X_f),X > 0,A = 1 \models X_f > 0)$. The subsequent sections will deal with the computation of summarizations.

6.2.3 Incremental Propagation of Strengthened Assertion

Suppose that we have a program Γ with clauses κ_1 to κ_n. By an application of LU, an assertion $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$ is unfolded into a number of assertions, each proved separately. There are five ways in which a child assertion is proved, represented by the proving of the following child assertions A_1 to A_5 :

1. Assertion $A_1 : \beta(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$ is obtained by a left unfold using constraint fact $\kappa_1 : p(\tilde{X})$:- $\beta(\tilde{X})$, and a direct proof using the assertion.

2. Assertion $A_2 : p(\tilde{X}'),\delta(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models H(\tilde{X})$ is obtained by left unfold using non-fact clause $\kappa_2 : p(\tilde{X})$:- $\delta(\tilde{X},\tilde{X}'),p(\tilde{X}')$ and is then proved by direct proof DP.

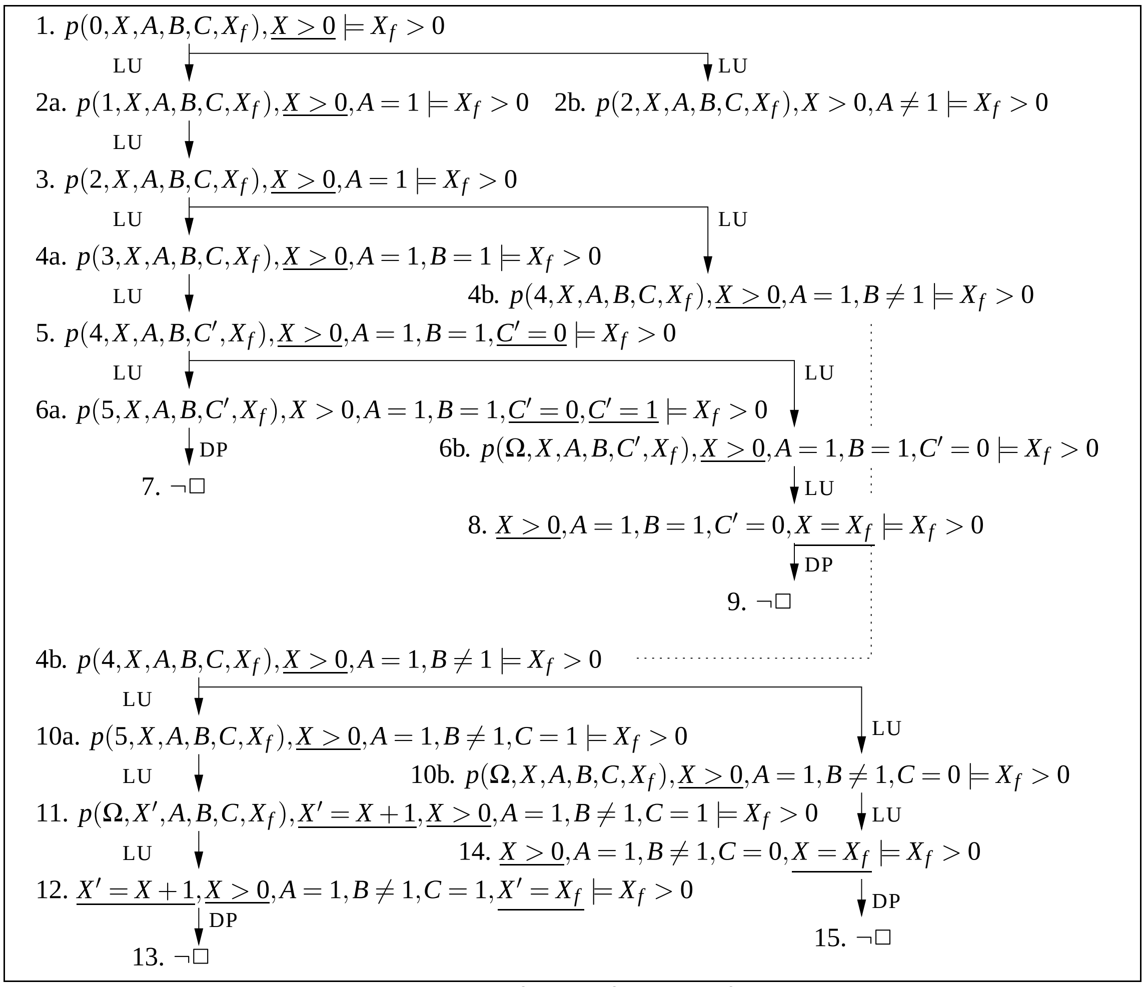
1. $p(0,X,A,B,C,X_f),\underline{X>0} \models X_f > 0$
LU
LU
2a. $p(1,X,A,B,C,X_f),\underline{X>0},A=1 \models X_f > 0$
2b. $p(2,X,A,B,C,X_f),X>0,A\neq 1 \models X_f > 0$
LU
3. $p(2,X,A,B,C,X_f),\underline{X>0},A=1 \models X_f > 0$
LU
LU
4a. $p(3,X,A,B,C,X_f),\underline{X>0},A=1,B=1 \models X_f > 0$
LU
4b. $p(4,X,A,B,C,X_f),\underline{X>0},A=1,B\neq 1 \models X_f > 0$
5. $p(4,X,A,B,C',X_f),\underline{X>0},A=1,B=1,\underline{C'=0} \models X_f > 0$
LU
LU
6a. $p(5,X,A,B,C',X_f),X>0,A=1,B=1,\underline{C'=0},\underline{C'=1} \models X_f > 0$
DP
6b. $p(\Omega,X,A,B,C',X_f),\underline{X>0},A=1,B=1,C'=0 \models X_f > 0$
7. $\neg\Box$
LU
8. $\underline{X>0},A=1,B=1,C'=0,\underline{X=X_f} \models X_f > 0$
DP
9. $\neg\Box$
4b. $p(4,X,A,B,C,X_f),\underline{X>0},A=1,B\neq 1 \models X_f > 0$
LU
LU
10a. $p(5,X,A,B,C,X_f),\underline{X>0},A=1,B\neq 1,C=1 \models X_f > 0$
LU
10b. $p(\Omega,X,A,B,C,X_f),\underline{X>0},A=1,B\neq 1,C=0 \models X_f > 0$
11. $p(\Omega,X',A,B,C,X_f),\underline{X'=X+1},\underline{X>0},A=1,B\neq 1,C=1 \models X_f > 0$
LU
LU
14. $\underline{X>0},A=1,B\neq 1,C=0,\underline{X=X_f} \models X_f > 0$
12. $\underline{X'=X+1},\underline{X>0},A=1,B\neq 1,C=1,\underline{X'=X_f} \models X_f > 0$
DP
DP
15. $\neg\Box$
13. $\neg\Box$

Figure 6.5: Optimized Proof Tree of Simple If Sequence Program

3. Assertion $A_3 : p(\tilde{X}'),\rho(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models H(\tilde{X})$ is obtained by left unfold using non-fact clause $\kappa_3 : p(\tilde{X})$:- $\rho(\tilde{X},\tilde{X}'),p(\tilde{X})$, and is then proved either by

   (a) further left unfold (LU),

   (b) applying induction hypothesis (AP), or,

   (c) application of CUT resulting in a proof of stronger assertion S where $S \rhd A_3$ (cf. Section 6.2.2).

Here, for example, the same unfold step using κ_1 can actually be used to prove a stronger assertion than the original $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$. For example, had the original assertion been $p(\tilde{X}) \models \beta(\tilde{X})$, we unfold this using the CLP clause κ_1 to $\beta(\tilde{X}) \models \beta(\tilde{X})$, which still holds. Notice that

$$(p(\tilde{X}) \models \beta(\tilde{X})) \rhd (p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X}))$$

because $\beta(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$. (That is, since $p(\tilde{X}) \models \beta(\tilde{X})$, we replace $\beta(\tilde{X})$ with $p(\tilde{X})$ resulting in $p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$.) In fact, $p(\tilde{X}) \models \beta(\tilde{X})$ is the strongest assertion which can be proved by an unfold step using κ_1.

We next demonstrate that for each unfold using κ_1, κ_2, or κ_3, there is a theoretical strongest assertion that can be proved. Due to the existence of the strongest assertion, the same unfold step can be used to prove anything weaker, including the original proved assertion. Since the original assertion may not be the strongest assertion, this opens the door to its strengthening.

Proposition 6.1. The strongest assertion that can be established by a left unfold step using a constraint fact $\kappa_1 : p(\tilde{X})$ `:-` $\beta(\tilde{X})$ is $p(\tilde{X}) \models \beta(\tilde{X})$.

Proof. Suppose that $p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$ is established by a left unfold using $p(\tilde{X})$ `:-` $\beta(\tilde{X})$, that is, $\beta(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$. Hence, $\beta(\tilde{X}) \Rightarrow (\phi(\tilde{X}) \Rightarrow H(\tilde{X}))$, and therefore $(p(\tilde{X}) \models \beta(\tilde{X})) \rhd (p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X}))$. $\Box$

The unfold using κ_2 is handled by the next proposition.

Proposition 6.2. The strongest assertion that can be directly proved after a left unfold using the clause $\kappa_2 : p(\tilde{X})$ `:-` $\delta(\tilde{X}, \tilde{X}'), p(\tilde{X}')$, where the result is proved using DP is $p(\tilde{X}) \models \delta(\tilde{X}, ?\tilde{X}'), p(?\tilde{X}')$.

Proof. Suppose that $p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$ is directly proved after a left unfold using the clause $p(\tilde{X})$ `:-` $\delta(\tilde{X}, \tilde{X}'), p(\tilde{X}')$. This means that $p(\tilde{X}'), \delta(\tilde{X}, \tilde{X}'), \phi(\tilde{X}) \models H(\tilde{X})$, hence $p(\tilde{X}'), \delta(\tilde{X}, \tilde{X}') \models (\phi(\tilde{X}) \Rightarrow H(\tilde{X}))$. Therefore, $(p(\tilde{X}) \models \delta(\tilde{X}, ?\tilde{X}'), p(?\tilde{X}')) \rhd (p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X}))$. $\Box$

The unfold using κ_3 is handled by the next proposition. We assume that the unfold result $A_3 : p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), \phi(\tilde{X}) \models H(\tilde{X})$ has been strengthened to assertion $S : p(\tilde{X}), \phi'(\tilde{X}) \models H'(\tilde{X}')$ and we want to know what is the strongest assertion which results in S after left unfold using the same clause κ_3.

Proposition 6.3. Suppose that an assertion $S : p(\tilde{X}'), \phi'(\tilde{X}') \models H'(\tilde{X}')$ has been established. The strongest assertion that is left unfolded to an assertion B where $S \rhd B$ using the clause $\kappa_3 : p(\tilde{X})$ `:-` $\rho(\tilde{X}, \tilde{X}'), p(\tilde{X}')$ is

$$p(\tilde{X}), \langle \forall \tilde{Y} : \rho(\tilde{X}, \tilde{Y}) \Rightarrow \phi'(\tilde{Y}) \rangle \models \rho(\tilde{X}, ?\tilde{Z}), H'(?\tilde{Z}) \tag{6.1}$$

Proof. First notice that (6.1) is equivalent to

$$\begin{aligned}
& p(\tilde{X}) \models \langle \forall \tilde{Y} : \rho(\tilde{X},\tilde{Y}) \Rightarrow \phi'(\tilde{Y}) \rangle \Rightarrow \langle \exists \tilde{Z} : \rho(\tilde{X},\tilde{Z}), H'(\tilde{Z}) \rangle \\
\equiv\ & p(\tilde{X}) \models \langle \exists \tilde{Y} : \neg(\neg\rho(\tilde{X},\tilde{Y}) \vee \phi'(\tilde{Y})) \rangle \vee \langle \exists \tilde{Z} : \rho(\tilde{X},\tilde{Z}), H'(\tilde{Z}) \rangle \\
\equiv\ & p(\tilde{X}) \models \langle \exists \tilde{Y} : \neg(\neg\rho(\tilde{X},\tilde{Y}) \vee \phi'(\tilde{Y})) \vee (\rho(\tilde{X},\tilde{Y}), H'(\tilde{Y})) \rangle \\
\equiv\ & p(\tilde{X}) \models \rho(\tilde{X}, ?\tilde{Y}), (\phi'(?\tilde{Y}) \Rightarrow H'(?\tilde{Y})).
\end{aligned}$$

Suppose that an assertion $p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$ is unfolded into assertion $B : p(\tilde{X}'), \rho(\tilde{X},\tilde{X}'), \phi(\tilde{X}) \models H(\tilde{X})$ using the clause κ_3.

Now, assertion $S \triangleright B$ holds. Then since S can be written as $p(\tilde{X}') \models (\phi'(\tilde{X}') \Rightarrow H'(\tilde{X}'))$, while B can be written as $p(\tilde{X}') \models (\rho(\tilde{X},\tilde{X}') \Rightarrow (\phi(\tilde{X}) \Rightarrow H(\tilde{X})))$, necessarily

$$\begin{aligned}
& (\phi'(\tilde{X}') \Rightarrow H'(\tilde{X}')) \Rightarrow (\rho(\tilde{X},\tilde{X}') \Rightarrow (\phi(\tilde{X}) \Rightarrow H(\tilde{X}))) \\
\equiv\ & (\rho(\tilde{X},\tilde{X}'), (\phi'(\tilde{X}') \Rightarrow H'(\tilde{X}'))) \Rightarrow (\phi(\tilde{X}) \Rightarrow H(\tilde{X})) \\
\equiv\ & \langle \exists \tilde{Y} : \rho(\tilde{X},\tilde{Y}), (\phi'(\tilde{Y}) \Rightarrow H'(\tilde{Y})) \rangle \Rightarrow (\phi(\tilde{X}) \Rightarrow H(\tilde{X})).
\end{aligned}$$

This means that (6.1) is stronger than $p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$. □

Case 3(a) is covered by Proposition 6.3 when S is a summarization of A_3, case 3(b) is covered when S is an already proved assertion, and case 3(c) is when S is a replacement assertion of A_3, where application of CUT establishes $S \triangleright A_3$.

When our CLP clauses represent the transitions of a state transition system, the computation of (6.1) from $A : p(\tilde{X}'), \phi'(\tilde{X}') \models H'(\tilde{X}')$ can be considered as:

- for the lhs of (6.1), computing a *weakest precondition* [47] of a transition relation $\rho(\tilde{x},\tilde{x}')$ represented as a CLP clause, and
- for the rhs of (6.1), a strongest postcondition (see Section 5.8.2) of its inverse transition.

We repeat here that $wp(t, \phi')$ is defined to be the most liberal condition, from which a transition step defined by the fragment t may reach a condition $\phi'(\tilde{x})$ on the program variables $\tilde{x}$. More formally, when ρ represents the transition relation defined by t [19],

$$wp(t, \phi') \equiv \langle \forall \tilde{x}' : \rho(\tilde{x},\tilde{x}') \Rightarrow \phi'(\tilde{x})\{\tilde{x} \mapsto \tilde{x}'\} \rangle.$$

Weakest precondition complements the strongest postcondition we have discussed in Section 5.8.1. Now, the strongest postcondition of an inverse ρ^{-1} of the transition relation ρ (that is, $\rho^{-1}(\tilde{x}',\tilde{x})$ if and only if $\rho(\tilde{x},\tilde{x}')$) defined by t, given the condition $H'(\tilde{x})$ is

$$sp(t^{-1},H') \equiv \langle \exists \tilde{x}' : \rho(\tilde{x},\tilde{x}') \wedge H'(\tilde{x})\{\tilde{x} \mapsto \tilde{x}'\}\rangle.$$

In the above, t^{-1} denotes the "inverse" or "backward" fragment defining the transition relation ρ^{-1}. From these, we can write (6.1) as

$$p(\tilde{X}), wp(t,\phi') \models sp(t^{-1},H').$$

As we have seen, left unfold using different CLP clauses result in different strengthening of the unfolded assertion $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$. Suppose that there are n clauses/unfolds, and therefore n strongest assertions $p(\tilde{X}),\phi_i(\tilde{X}) \models H_i(\tilde{X})$ for $1 \leq i \leq n$. The strongest assertion is in general not computable and often inefficient to compute. Hence instead of the strongest assertion $p(\tilde{X}),\phi_i(\tilde{X}) \models H_i(\tilde{X})$, we compute a weaker $p(\tilde{X}),\phi_i'(\tilde{X}) \models H_i'(\tilde{X})$ where $(p(\tilde{X}),\phi_i(\tilde{X}) \models H_i(\tilde{X})) \rhd (p(\tilde{X}),\phi_i'(\tilde{X}) \models H_i'(\tilde{X}))$. However, in order to still be useful as a strengthening of the unfolded assertion, we require that $(p(\tilde{X}),\phi_i'(\tilde{X}) \models H_i'(\tilde{X})) \rhd (p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X}))$. That is, it is actually stronger than the original assertion.

Based on our discussions so far, the strengthening of the unfolded assertion $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$ can now be written as the disjunction

$$\bigvee_{i=1}^{n} (p(\tilde{X}),\phi_i'(\tilde{X}) \models H_i'(\tilde{X})). \tag{6.2}$$

Recall that our purpose in computing the strengthening of the unfolded assertion $p(\tilde{X}),\phi(\tilde{X}) \models H(\tilde{X})$ is such that its proof is more likely to make the proving of other assertions redundant. Therefore we need to memo (6.2) in computer memory. However, storing a disjunction of assertions can be inefficient (it can be exponential to the depth of the subtree). We therefore propose to store a single assertion instead.

In the following proposition, we state how we may construct a single summarization of the original obligation out of the stronger assertions returned by the unfold children.

Proposition 6.4. Suppose that we managed to prove an assertion $A : p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$ by first applying LU using all the $\kappa_1, \ldots, \kappa_n$ clauses of a CLP program. Now, $p(\tilde{X}), \phi_i'(\tilde{X}) \models H_i'(\tilde{X})$ is an assertion which can be proved by an unfold using clause κ_i only, for $1 \leq i \leq n$. An assertion $S : p(\tilde{X}), \phi''(\tilde{X}) \models H''(\tilde{X})$ such that

- $\phi''(\tilde{X}) \Rightarrow \bigwedge_{i=1}^{n} \phi_i'(\tilde{X})$,
- $H''(\tilde{X}) \Leftarrow \bigvee_{i=1}^{n} H_i'(\tilde{X})$, and
- $S \rhd A$,

is a summarization of A.

Proof. The condition that $S \rhd A$ for summarization is satisfied by definition. Now we demonstrate that the proof tree of A can be used to prove S also. Note that for any $1 \leq i \leq n$, $(p(\tilde{X}), \phi_i'(\tilde{X}) \models H_i'(\tilde{X})) \rhd S$. This is because $\phi''(\tilde{X}) \Rightarrow \phi_i'(\tilde{X})$ and $H_i'(\tilde{X}) \Rightarrow H''(\tilde{X})$ for all $1 \leq i \leq n$. Therefore S can be proved by left unfold using any rule from κ_1 to κ_n. $\Box$

In the next section we formalize the computing of summarizations via constraint deletion, which has been exemplified in Section 6.2.1.

6.2.4 Constraint Deletion

Deletion Functions

Here we first propose black-box primitives of an efficient proof algorithm which employs constraint deletion to strengthen assertions. An essential element of the algorithm is the storing and manipulation of sequence of constraints corresponding to the unfold path.

We first define a function $cdel(A, C)$ which, when given an assertion A and a clause C, computes a stronger assertion than A using constraint deletion technique, based on the information obtained by left unfold using clause C. Here we also consider the three cases mentioned in Section 6.2.3.

For the unfold using constraint fact κ_1, the value of $cdel(A, C) = cdel_1(A, C)$, where the value of $cdel_1$ is given as follows:

$$cdel_1((p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H), \kappa_1) =$$
$$\begin{cases} p(\tilde{X}), c_{a+1}, \ldots, c_b \models \Box \text{ if } \beta(\tilde{X}), c_{a+1}, \ldots, c_b \models \Box \\ p(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H} \text{ if } \beta(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H} \text{ and } \check{H}, c_1, \ldots, c_a \models H. \end{cases} \tag{6.3}$$

Note that in this definition we do not specify how to compute the goal $\check{H}$. This will be the subject of the next section. In particular, $\check{H}$ can be simply H, as in the proof of our first example in Section 6.2.1.

Before discussing the *cdel* for the second case, we first introduce a restriction that we only consider left unfold using κ_2 where the lhs of the resulting A_2 evaluates to $\Box$. That is, $p(\tilde{X}'), \delta(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models H(\tilde{X})$ is directly proved by our algorithm only when $\delta(\tilde{X},\tilde{X}'),\phi(\tilde{X}) \models \Box$.

The value of $cdel(A,C)$ for this case is given by the function $cdel_2(A,C)$ defined as follows.

$$\begin{aligned}&cdel_2((p(\tilde{X}),c_1,\ldots,c_a,c_{a+1},\ldots,c_b \models H),\kappa_2) = \\ &\quad p(\tilde{X}),c_{a+1},\ldots,c_b \models \Box \text{ where } \delta(\tilde{X},\tilde{X}'),c_{a+1},\ldots,c_b \models \Box\end{aligned} \tag{6.4}$$

We now consider cases 3(a), 3(b), and 3(c) separately.

For case 3(a), where A_3 is proved by further left unfold, we define the following $cdel_{3(a)}$ function:

$$\begin{aligned}&cdel_{3(a)}((p(\tilde{X}),c_1,\ldots,c_a,c_{a+1},\ldots,c_b \models H),\kappa_{r.i}) = \\ &\quad p(\tilde{X}),c_{a+1},\ldots,c_b \models \check{H} \text{ where} \\ &\qquad 1.\ \ summ(p(\tilde{X}'),\rho_i(\tilde{X},\tilde{X}'),c_1,\ldots,c_b \models H) = p(\tilde{X}'),e_1,\ldots,e_d \models H'(\tilde{X}') \\ &\qquad\quad\ \text{where } \{c_1,\ldots,c_a\} \cap \{e_1,\ldots,e_d\} = \emptyset, \\ &\qquad 2.\ \ \rho_i(\tilde{X},\tilde{X}'),H'(\tilde{X}') \models \check{H},\ \text{ and} \\ &\qquad 3.\ \ \check{H},c_1,\ldots,c_a \models H.\end{aligned} \tag{6.5}$$

The function $cdel_{3(a)}$ above uses the function *summ* which we will define later. For now, it suffices to say that $summ(p(\tilde{X}'),\rho_i(\tilde{X},\tilde{X}'),c_1,\ldots,c_b \models H)$ is only defined when its argument $p(\tilde{X}'),\rho_i(\tilde{X},\tilde{X}'),c_1,\ldots,c_b \models H$ is provable starting with a left unfold (LU), and it returns a stronger assertion $p(\tilde{X}),e_1,\ldots,e_d \models H'$ which can also be proved given the CLP program Γ, such that $\{e_1,\ldots,e_d\} \subseteq \{\rho_i(\tilde{X},\tilde{X}'),c_1,\ldots,c_b\}$ and $H' \models H$.

We now discuss the case 3(b) where the value of *cdel* is given by the function $cdel_{3(b)}$ as follows.

$$
\begin{aligned}
& cdel_{3(\mathrm{b})}((p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H), \kappa_3) = \\
& \quad p(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H} \text{ where} \\
& \qquad 1. \ \text{an assertion } S : p(\tilde{Y}), \phi \models H'(\tilde{Y}) \text{ is proved or assumed and} \\
& \qquad \ \theta \text{ renames away } S \text{ from } p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_1, \ldots, c_b \models H, \\
& \qquad 2. \ \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \phi\theta, \tilde{X}' = \tilde{Y} \ \text{ (subsumption)}, \\
& \qquad 3. \ \rho(\tilde{X}, \tilde{X}'), H'(\tilde{Y})\theta, \tilde{X}' = \tilde{Y} \models \check{H} \ \text{ (residual assertion), and} \\
& \qquad 4. \ \check{H}, c_1, \ldots, c_a \models H.
\end{aligned}
\tag{6.6}
$$

We now consider case 3(c). Here the value of $cdel(A,C)$ is given by $cdel_{3(\mathrm{c})}(A,C)$ as follows.

$$
\begin{aligned}
& cdel_{3(\mathrm{c})}((p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H), \kappa_3) = \\
& \quad p(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H} \text{ where} \\
& \qquad 1. \ \text{an assertion } S : G_S \models H_S \text{ is proved,} \\
& \qquad \ \theta \text{ renames away } S \text{ from } A_3 : p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_1, \ldots, c_b \models H, \text{ and} \\
& \qquad \ \tilde{Y} \subseteq var(S\theta) \text{ and } \tilde{Z} \subseteq var(A_3) \text{ s.t. } |\tilde{Y}| = |\tilde{Z}|. \\
& \qquad 2. \ p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models G_S\theta, \tilde{Y} = \tilde{Z} \ \text{ (subsumption)}, \\
& \qquad 3. \ \rho(\tilde{X}, \tilde{X}'), H_S\theta, \tilde{Y} = \tilde{Z} \models \check{H} \ \text{ (residual assertion), and} \\
& \qquad 4. \ \check{H}, c_1, \ldots, c_a \models H.
\end{aligned}
\tag{6.7}
$$

$cdel_{3(\mathrm{c})}$ is similar to $cdel_{3(\mathrm{b})}$. The difference is only in the way subsumption test is done.

We now define the function $summ(A)$, given an assertion A. The function is only defined when A is provable using one application of LU, followed by arbitrary applications of LU, DP, AP, and RU, and following our restriction mentioned above, that we never use DP to conclude a proof obligation unless either the lhs of the assertion evaluates to $\Box$ or the assertion is obtained by a left unfold using a constraint fact.

Suppose that $A : p(\tilde{X}), \phi(\tilde{X}) \models H(\tilde{X})$ is completely unfolded by applying RU rule using all clauses $\kappa_1, \ldots, \kappa_n$ of a CLP program. Suppose that $cdel(A, \kappa_i)$ is $p(\tilde{X}), \hat{\phi}_i(\tilde{X}) \models \check{H}_i(\tilde{X})$ for $1 \leq i \leq n$. Following from our discussions above, we know that $\hat{\phi}_i(\tilde{X})$ can be viewed as a set of constraints which is a subset of $\phi(\tilde{X})$.

We define:

$$summ(A) \equiv p(\tilde{X}), \bigcup_{i=1}^{n} \hat{\phi}_i(\tilde{X}) \models \check{H}(\tilde{X}), \tag{6.8}$$

where $\check{H}(\tilde{X}) \Leftarrow \bigvee_{i=1}^{n} \check{H}_i(\tilde{X})$ and $\check{H}(\tilde{X}) \models H(\tilde{X})$. Note that again here we leave open how we may compute $\check{H}$.

Correctness

We will now prove the *correctness* and *usefulness* lemmas of the functions defined above. Later in this section we will establish that $summ(A)$ is a summarization of A (according to Definition 6.1 on Page 173).

Definiton 6.2 (Correctness). $Cdel_{\mathcal{X}}(A,\kappa)$ is *correct* for either $\mathcal{X} = 1$, 2, 3(a), 3(b), or 3(c) when it can be established by left unfolding using the clause κ.

Definiton 6.3 (Usefulness). $Cdel_{\mathcal{X}}(A,\kappa)$ is *useful* for either $\mathcal{X} = 1$, 2, 3(a), 3(b), or 3(c) when $cdel_{\mathcal{X}}(A,\kappa) \rhd A$.

Lemma 6.1. $Cdel_1(A,\kappa_1)$ is correct and useful.

Proof. Suppose that A is $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H$ and $cdel_1(A,\kappa_1)$ is given as in (6.3). We consider two cases:

1. $\beta(\tilde{X}), c_{a+1}, \ldots, c_b \models \Box$. In this case, $cdel_1(A,\kappa_1) = p(\tilde{X}), c_{a+1}, \ldots, c_b \models \Box$. Since this can be established by a left unfold step using κ_1 then necessarily $(p(\tilde{X}) \models \beta(\tilde{X})) \rhd cdel_1(A,\kappa_1)$. In order to prove the usefulness, we assume that $p(\tilde{X}), c_{a+1}, \ldots, c_b \models \Box$, i.e., $cdel_1(A,\kappa_1)$ holds, and we prove A. It is easy to see that

   (a) $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models p(\tilde{X}), c_{a+1}, \ldots, c_b$.

   (b) $\Box \models H$.

   Combining the above 2 with $cdel_1(A,\kappa_1)$ by modus ponens we establish A.

2. $\beta(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H}$ and $\check{H} \models H$. In this case, $cdel_1(A,\kappa_1) = p(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H}$ is correct since it can be proved by left unfold using clause κ_1, therefore necessarily $(p(\tilde{X}) \models \beta(\tilde{X})) \rhd cdel_1(A,\kappa_1)$. For the usefulness, we assume that $cdel_1(A,\kappa_1)$ holds and we derive A. Here it is easy to see from $cdel_1(A,\kappa_1)$ that

$$p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models \check{H}, c_1, \ldots, c_a.$$

Since $\check{H}, c_1, \ldots, c_a \models H$, we get A. This establishes $cdel_1(A, \kappa_1) \rhd A$. $\Box$

Lemma 6.2. *$Cdel_2(A, \kappa_2)$ is correct and useful.*

Proof. Suppose that A is $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H$, and $cdel_2(A, \kappa_2) = p(\tilde{X}), c_{a+1}, \ldots,$ $c_b \models \Box$ where $\delta(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \Box$. $Cdel(A, \kappa_2)$ is correct because by left unfolding it using κ_2 we get $p(\tilde{X}'), \delta(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \Box$ which holds since $\delta(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \Box$.

Also, $cdel_2(A, \kappa_2)$ is useful since, first of all it is easy to see that

1. $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models p(\tilde{X}), c_{a+1}, \ldots, c_b$.

2. $\Box \models H$.

Now, using $cdel_2(A, \kappa_2)$ we derive A by modus ponens using the above 2 assertions. $\Box$

Lemma 6.3. *$Cdel_{3(\mathrm{a})}(A, \kappa_3)$ is correct and useful.*

Proof. Suppose that A is $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H$ and $cdel_{3(\mathrm{a})}(A, \kappa_3) = p(\tilde{X}), c_{a+1}, \ldots,$ $c_b \models \check{H}$. We now left unfold $cdel_{3(\mathrm{a})}(A, \kappa_3)$ using κ_3 resulting in $p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models$ $\check{H}$. Since $\{c_1, \ldots, c_a\} \cap \{e_1, \ldots, e_d\} = \emptyset$ then necessarily $\{e_1, \ldots, e_d\} \subseteq \{\rho_i(\tilde{X}, \tilde{X}'), c_1, \ldots, c_b\}$, therefore $p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models p(\tilde{X}'), e_1, \ldots, e_d$, and therefore $p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), c_{a+1},$ $\ldots, c_b \models p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), e_1, \ldots, e_d$. Now since $p(\tilde{X}'), e_1, \ldots, e_d \models H'(\tilde{X}')$ we have that $p(\tilde{X}'),$ $\rho_i(\tilde{X}, \tilde{X}'), e_1, \ldots, e_d \models \rho(\tilde{X}, \tilde{X}'), H'(\tilde{X}')$. By modus ponens we establish $p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), c_{a+1},$ $\ldots, c_b \models \rho(\tilde{X}, \tilde{X}'), H'(\tilde{X}')$, and since $\rho(\tilde{X}, \tilde{X}'), H'(\tilde{X}') \models \check{H}$, again by modus ponens we finally establish $p(\tilde{X}'), \rho_i(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \check{H}$.

On the usefulness, it is easy to see that from $cdel_{3(\mathrm{a})}(A, \kappa_3)$ we have that $p(\tilde{X}), c_1, \ldots, c_a,$ $c_{a+1}, \ldots, c_b \models \check{H}, c_1, \ldots, c_a$. And since $\check{H}, c_1, \ldots, c_a \models H$, we derive A, and therefore $cdel_{3(\mathrm{a})}(A, \kappa_3)$ $\rhd A$. $\Box$

Lemma 6.4. *$Cdel_{3(\mathrm{b})}(A, \kappa_3)$ is correct and useful.*

Proof. Suppose that A is $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H$ and $cdel_{3(\mathrm{b})}(A, \kappa_3) = (p(\tilde{X}), c_{a+1}, \ldots,$ $c_b \models \check{H})$. Unfolding $cdel_{3(\mathrm{b})}(A, \kappa_3)$ we get $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \check{H}$. Due to condition no. 2 in (6.6), we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), \phi\theta, \tilde{X}' = \tilde{Y}$. By modus ponens using assertion S we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \rho(\tilde{X}, \tilde{X}'), H'(\tilde{Y})\theta, \tilde{X}' = \tilde{Y}$. By another modus ponens using condition no. 3, we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \check{H}$, which is the left unfold of $cdel_{3(b)}(A, \kappa_3)$.

For the usefulness, it is easy to see from $cdel_{3(b)}(A, \kappa_3)$ that $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models \check{H}, c_1, \ldots, c_a$. And since $\check{H}, c_1, \ldots c_a \models H$, therefore A holds, and therefore $cdel_{3(b)}(A, \kappa_3) \rhd A$. □

Lemma 6.5. $cdel_{3(c)}(A, \kappa_3)$ is correct and useful.

Proof. The proof is similar to the proof of Proposition 6.4.

Suppose that A is $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models H$ and $cdel_{3(c)}(A, \kappa_3) = (p(\tilde{X}), c_{a+1}, \ldots, c_b \models \check{H})$. Unfolding $cdel_{3(c)}(A, \kappa_3)$ we get $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \check{H}$. Due to condition no. 2 in (6.7), we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), G_S\theta, \tilde{Y} = \tilde{Z}$. By modus ponens using assertion S we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \rho(\tilde{X}, \tilde{X}'), H_S\theta, \tilde{Y} = \tilde{Z}$. By another modus ponens using condition no. 3, we have that $p(\tilde{X}'), \rho(\tilde{X}, \tilde{X}'), c_{a+1}, \ldots, c_b \models \check{H}$, which is the left unfold of $cdel_{3(c)}(A, \kappa_3)$.

For the usefulness, it is easy to see from $cdel_{3(c)}(A, \kappa_3)$ that $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots, c_b \models \check{H}, c_1, \ldots, c_a$. And since $\check{H}, c_1, \ldots, c_a \models H$, we derive A, and therefore $cdel_{3(c)}(A, \kappa_3) \rhd A$. □

Theorem 6.1 (Summarization by Deletion). $Summ(A)$ is a summarization of A.

Proof. From Definition 6.1, here we need to establish the following two:

1. That $Summ(A)$ can be proved by the same left unfold as is applied to A (and hence proved by the same subtree.

2. That $Summ(A) \rhd A$.

From Lemmas 6.1, 6.2, 6.3, and 6.4, we know that $cdel(A, \kappa_i)$ is correct for $1 \leq i \leq n$. Now, for any $1 \leq i \leq n$, obviously $p(\tilde{X}), \bigcup_{j=1}^{n} \hat{\phi}_j(\tilde{X}) \models p(\tilde{X}), \hat{\phi}_i(\tilde{X})$. Moreover, $\check{H}_i(\tilde{X}) \Rightarrow \check{H}(\tilde{X})$. Hence, for all $1 \leq i \leq n$, $cdel(A, \kappa_i) \rhd summ(A)$. Since $cdel(A, \kappa_i)$ is correct for all $1 \leq i \leq n$, then $summ(A)$ can be proved by the same left unfold using $\kappa_1, \ldots, \kappa_n$.

We now prove $summ(A) \rhd A$. We now demonstrate how we may obtain A from $summ(A)$. Obviously $p(\tilde{X}), \phi(\tilde{X}) \models p(\tilde{X}), \bigcup_{i=1}^{n} \hat{\phi}_i(\tilde{X})$ since for all $1 \leq i \leq n$, $\hat{\phi}_i(\tilde{X}) \subseteq \phi(\tilde{X})$. Using this and $summ(A)$ and by modus ponens we have that $p(\tilde{X}), \phi(\tilde{X}) \models \check{H}(\tilde{X})$, and since $\check{H}(\tilde{X}) \models H(\tilde{X})$, we obtain A. □

6.2.5 Information Discovery via Dynamic Summarization

In the previous section we have described the constraint deletion technique to generalize lhs of an assertion in order to obtain a candidate for summarization, which is more informative than the original assertion. Generalization of the lhs can be coupled with a specialization of the rhs of an assertion for the same purpose, and this is the focus of this section.

Also in the previous section we have assumed the existence of a function which would automatically produce $\check{H}$ from an unfold of $p(\tilde{X}), c_1, \ldots, c_a, c_{a+1}, \ldots c_b \models H$ such that $\check{H}, c_1, \ldots, c_a \models H$. We can view $\check{H}, c_1, \ldots, c_a$ as a specialization of H. In normal safety proof, as in our example in Section 6.2.1, $\check{H}$ is simply H, but we can be more flexible depending on the given problem. One such problem is the proof of execution time bound, which we will use as our main example. The proof will become essentially a discovery process of the time bound.

Users may initially guide the information that they want to extract by providing an initial lhs H with existentially quantified variables. For example, when we wish to discover the timing bound of a program, we run a dynamic summarization-based algorithm with input the assertion

$$p(0, \tilde{X}, T, T_f), \phi(\tilde{X}), T = 0 \models T_f - T \leq ?Bound.$$

Here, T represents the current execution time of the program, and T_f is a final variable representing the execution time at the end of the program. We assume that the program indeed terminates, and there is an actual value for B.

Now suppose that a sequence of left unfolds updates the above assertion into the following assertion A at depth k (assuming the increment of variable T is 1 with each unfold), which is to be unfolded using a constraint fact:

$$p(l, \tilde{X}^k, T^k, T_f), \phi(\tilde{X}), \ldots, T = 0, T' = T + 1, T'' = T' + 1, \ldots, T^k = T^{k-1} + 1 \models T_f - T \leq ?Bound.$$

Again suppose that in the program we have the constraint fact κ_1 :

$$p(l, \tilde{X}, T, T_f) \;\texttt{:-}\; T = T_f.$$

The result of left unfold of A using this clause is the assertion

$$T_k = T_f, \phi(\tilde{X}), \ldots, T' = T + 1, T'' = T' + 1, \ldots, T^k = T^{k-1} + 1 \models T_f - T \leq ?Bound.$$

Computing the function $cdel_1(A,\kappa_1)$ where $\check{H} = H$ results in the following assertion A' :

$$p(l,\tilde{X}^k,T^k,T_f), T' = T+1, T'' = T'+1, \ldots, T^k = T^{k-1}+1 \models T_f - T \leq ?Bound.$$

Here, the constraints sequence $T = 0, \phi(\tilde{X}), \ldots,$ in A is not necessary for the assertion to hold, hence they are deleted from A. The assertion A' is already a correct candidate for summarization of A, however, we can produce more informative candidate by the following procedure:

- We first produce $\check{H}$ such that $\check{H}, T' = T+1, T'' = T'+1, \ldots, T^k = T^{k-1}+1 \models T_f - T \leq ?Bound$. In the special case of proving timing bound, $\check{H}$ can be easily determined to be the bound $T_f - T^k \leq 0$ between T_f and T_k.
- We then examine the assertion A' with its rhs replaced with $T_f - T^k \leq 0$:

  $$p(l,\tilde{X}^k,T^k,T_f), T' = T+1, T'' = T'+1, \ldots, T^k = T^{k-1}+1 \models T_f - T^k \leq 0.$$

  Now, the constraints $T' = T+1, T'' = T'+1, \ldots, T^k = T^{k-1}+1$ are no longer necessary to imply the rhs, and can be further removed, resulting in our final candidate A'' :

  $$p(l,\tilde{X}^k,T^k,T_f) \models T_f - T^k \leq 0.$$

  It is easy to see that A'' satisfies all the properties of $cdel_1(A,\kappa_1)$ (6.3), and it is more informative than A', i.e., $A'' \rhd A'$.

We now deal with the question of propagating the candidate summarization to the ancestors. Let us consider $cdel_{3(\mathrm{a})}$ (cases $cdel_{3(\mathrm{b})}$ and $cdel_{3(\mathrm{c})}$ are similar). Consider an immediate ancestor of A, which we call B :

$$p(m,\tilde{X}^{k-1},T^{k-1},T_f), \phi(\tilde{X}), \ldots, T' = T+1, T'' = T'+1, \ldots, T^{k-1} = T^{k-2}+1 \models T_f - T \leq ?Bound.$$

Here we want to generate a candidate summarization of B from A''. Suppose that B is unfolded to A by the clause κ_2 :

$$p(m,\tilde{X},T,T_f) \ :\!-\ \tilde{X} = \tilde{X}', T' = T+1, p(l,\tilde{X},T',T_f).$$

First we consider the lhs of $cdel_{3(\mathrm{a})}(B,\kappa_2)$. Since A'' does not require all constraints in $\phi(\tilde{X}), \ldots,$

$T' = T+1, T'' = T'+1, \ldots, T^k = T^{k-1}+1$, we can delete all of these from B. We next consider the rhs of $cdel_{3(a)}(B, \kappa_2)$. Again here we want to produce a $\check{H}$, but now two conditions must hold (conditions 2 and 3 of (6.5)):

1. $\tilde{X}^{k-1} = \tilde{X}^k, T^k = T^{k-1}+1, T_f - T^k \leq 0 \models \check{H}$, and
2. $\check{H}, \phi(\tilde{X}), \ldots, T' = T+1, T'' = T'+1, \ldots, T^{k-1} = T^{k-2}+1 \models T_f - T \leq ?Bound$.

For our special problem, it is easy to determine such $\check{H}$ to be $T_f - T^{k-1} \leq 1$. Hence, we have a candidate summarization B_1 :

$$p(m, \tilde{X}^{k-1}, T^{k-1}, T_f) \models T_f - T^{k-1} \leq 1.$$

The remaining problem is on computing a single assertion which is a summarization of B, i.e., $summ(B)$ according to (6.8). Suppose that B can be unfolded using another clause κ_3, which produces a candidate summarization B_2 as follows:

$$p(m, \tilde{X}^{k-1}, T^{k-1}, T_f) \models T_f - T^{k-1} \leq 2.$$

Both B_1 and B_2 have the same lhs. One of the correct $summ(B)$ is

$$p(m, \tilde{X}^{k-1}, T^{k-1}, T_f) \models T_f - T^{k-1} \leq 2 \vee T_f - T^{k-1} \leq 1.$$

Unfortunately, in this case the rhs contains a disjunction, which can be of exponential size as summarizations are propagated to the ancestors. From Proposition 6.4, however, we know that it is enough to use an expression which is a cover of the disjunction. In our special case of timing bound discovery, this is always one of the disjuncts, which in the above case is $T_f - T^{k-1} \leq 2$. We therefore obtain $summ(B) = B_2$. As it is here, in its most generality, we derive a disjunction at the rhs, but it can often be simplified.

We note that although, this technique discovers bounds, it does not discover the tightest bound. For instance, the most that we can guarantee is that it discovers some timing bound and not the *worst-case execution time* (*WCET*). The problem is that when we consider an assertion to be proved is redundant to a summarization stored in the table, it is not necessarily the case that the path that gives rise to the timing bound of the summarization is also feasible in the redundant assertion. As the result of redundancy test, we may get an answer bound for the redundant

```
      proc $summarize(G \models H)$
⟨0⟩     $Table^L, NotDone^L := \emptyset, \emptyset$
⟨1⟩     $Summarization := recurse\_summ(\emptyset, G \models H)$
⟨2⟩     return $Summarization, NotDone^L$
      end proc

      proc $recurse\_summ(\tilde{A}^L, G \models H)$
⟨3⟩     $L, R := \top, \bot$
⟨4⟩     for each $(\kappa \in \Gamma)$ do
⟨5⟩       $g := unfold^{\kappa}(G)$
⟨6⟩       if $(abstraction\_point(g))$ then
⟨7⟩         if $(S \in Table$ s.t. $S \rhd_p (g \models H))$ then
⟨8⟩           $(G' \models H') := cdel_{3(b)}(G \models H, \kappa)$
            else
⟨9⟩           choose $S$ s.t. $S \rhd (g \models H)$
⟨10⟩          $NotDone^L := NotDone^L \cup \{S\}$
⟨11⟩          $(G' \models H') := cdel_{3(c)}(G \models H, \kappa)$
            end if
⟨12⟩      else if $(S \in \tilde{A}^L$ s.t. $S \rhd_p (g \models H))$ then $(G' \models H') := cdel_{3(b)}(G \models H, \kappa)$
⟨13⟩      else if $(S \in Table^L$ s.t. $S \rhd_p (g \models H))$ then $(G' \models H') := cdel_{3(b)}(G \models H, \kappa)$
⟨14⟩      else if $(\kappa$ is non-fact and $g$ is $\Box)$ then $(G' \models H') := cdel_2(G \models H, \kappa)$
⟨15⟩      else if $(\kappa$ is fact$)$ then
⟨16⟩        if $(g \models H$ is provable$)$ then $(G' \models H') := cdel_1(G \models H, \kappa)$
⟨17⟩        else abort end if
          else
⟨18⟩        $(G' \models H') := recurse\_summ(\tilde{A}^L \cup \{G \models H\}, g \models H)$
          end if
⟨19⟩      $L, R := intersect(L, G'), closure(R, H')$
        end for
⟨20⟩    $Table^L := Table^L \cup \{L \models R\}$
⟨21⟩    return $(L \models R)$
      end proc
```

Figure 6.6: *Summarize* Procedure

assertion which is actually an over approximation.

Other than the discovery of timing bound or bounds on resource usage in general, the technique explained in this section can potentially be extended for retrieving various information about a program.

6.3 The Basic Compositional Algorithm

Before we present our main algorithm, we first provide an algorithm which performs traversal with dynamic summarization, in the form of *summarize* procedure shown in Figure 6.6. *Summarize* is simply a wrapper to the procedure *recurse_summ*. The procedure accesses two

```
         proc $prove(G \models H)$
⟨0⟩        $Table, NotDone := \emptyset, \emptyset$
⟨1⟩        $(G' \models H'), NotDone' := summarize(G \models H)$
⟨2⟩        $Table := Table \cup \{G' \models H'\}$
⟨3⟩        $NotDone := NotDone \cup NotDone'$
⟨4⟩        while $(NotDone \neq \emptyset)$ do
⟨5⟩          $(G \models H) := pop(NotDone)$
⟨6⟩          $(G' \models H'), NotDone' := summarize(G \models H)$
⟨7⟩          $Table := Table \cup \{G' \models H'\}$
⟨8⟩          $NotDone := NotDone \cup NotDone'$
           end do
           return $Success$
         end proc
```

Figure 6.7: Compositional Algorithm

kinds of table represented as the two variables *Table* and $Table^L$. *Table* is defined in the main program (to be shown later), and is a global table for the whole program. It stores proved assertions *at abstraction points*. On the other hand, $Table^L$ is the local table for the current execution of *summarize*, where strengthening of proved assertions which are not abstraction points are stored.

We now explain *recurse_summ* in more detail. *Recursesumm* is called with two arguments: $\tilde{A}^L$ represents the current assumed assertions to be used in coinduction, and $G \models H$ is the assertion which is to be proved and its strengthening produced. At ⟨3⟩ we reset the two variables L and R. L stores the current computed lhs of the summarization, while R stores the rhs. L is initialized to $\top$, while R is initialized to $\perp$. $L \models R$ is eventually to be returned to the caller, and its final value is computed incrementally.

The algorithm basically attempts to perform left unfold using each CLP clause κ in the program Γ and update the summarization $L \models R$. To denote an unfold using a particular κ, here we define a function $unfold^\kappa$, such that $unfold^\kappa_{p(\tilde{X})}(G) = G'$ when $\square$:- $G' \equiv resolv_{p(\tilde{X})}(\square$:- $G, \kappa)$. Note that $unfold_{p(\tilde{X})}(G) = \{unfold^\kappa_{p(\tilde{X})}(G) | \kappa \in \Gamma\}$. We simply write $unfold^\kappa_{p(\tilde{X})}$ as $unfold^\kappa$ when the unfolded atom $p(\tilde{X})$ is clear.

After unfolding using κ, the procedure may encounter abstraction points (the condition at ⟨5⟩ holds). Here there are two cases: the summarization of the abstraction point is already kept in the global table, in which case we generate a summarization using $cdel_{3(\mathrm{b})}$. Otherwise, we include some stronger assertion S in $NotDone^L$. $NotDone^L$ is a set which stores all encountered abstraction points whose assertions are not yet proved. It is to be passed to the main program.

The procedure may also encounter a point where we apply coinduction (Line ⟨11⟩). It is also possible that the unfolded assertion is redundant to a summarization in the *Table*^{L} (Line $\langle 12 \rangle$). Line $\langle 13 \rangle$ handles the case when the κ is not a fact, yet g evaluates to $\Box$. Line $\langle 15 \rangle$–$\langle 16 \rangle$ handle the case when κ is a fact. If none of the above cases apply, we have not yet been able to produce a summarization, in which case we need to go deeper in the proof tree by recursing on *recurse_summ* (Line $\langle 17 \rangle$).

At Line $\langle 18 \rangle$ we generate $L \models R$ incrementally using 2 procedures: *intersect* and *closure*. The function $\mathit{intersect}(L, G')$ computes some condition φ such that $\varphi \Rightarrow (L \wedge G')$. In the case of constraint deletion, $\mathit{intersect}(L, G')$ simply returns the union of the conjuncts in L and G. On the other hand, $\mathit{closure}(R, H')$ computes a condition ψ such that $(R \vee H') \Rightarrow \psi$. We may adopt an implementation where in case R is $\Box$ then $\mathit{closure}(R, H')$ is H' and similarly when H' is $\Box$ then $\mathit{closure}(R, H')$ is R. In case we attempt to discover timing bounds, when R is the constraint $T_f - T \leq \alpha$ and H' is the constraint $T_f - T \leq \beta$, $\mathit{closure}(R, H') \equiv T_f - T \leq \mathit{max}(\alpha, \beta)$. In case we are proving a safety condition and neither R nor H' is $\Box$, then $R \equiv H' \equiv \mathit{closure}(R, H')$. In this way we have an iterative computation of $\mathit{summ}(G \models H)$, whose formal definition we give in Section 6.2.4.

At Line $\langle 19 \rangle$ the procedure updates the local table by adding the new summarization $L \models R$ and returns the assertion at Line $\langle 20 \rangle$.

We show the pseudocode of our main algorithm in Figure 6.7. The algorithm calls the procedure $\mathit{summarize}(G \models H)$ which returns a strengthening of $G' \models H'$ of $G \models H$ and a set of assertions that are not yet proved in the set *NotDone*$'$, which is then added to the set *NotDone*. The program iterates until *NotDone* is an empty set.

We now exemplify a compositional proof using our algorithm. Here we return to our example Program 6.1, which has been proved compositionally in Section 5.8.4, and proved using dynamic summarization in Section 6.2.1. Here we again prove the example, but using both composition and dynamic summarization.

We show the proof tree in Figure 6.8. We also define $\langle 4 \rangle$ as a breakpoint of two fragments as in Section 5.8.4, but now we reason on each fragment using dynamic summarization. Here, we assume that the user have specified that any assertion $G \models H$ whose lhs satisfies $p(4, X, A, B, C, X_f)$, $X > 0$ is an abstraction point, and that $p(4, X, A, B, C, X_f), X > 0 \models X_f > 0$ needs to be proved. We start the execution of our algorithm by the call

$$\mathit{prove}(p(0, X, A, B, C, X_f), X > 0 \models X_f > 0).$$

$$\begin{tikzcd}
1.\ p(4,X,A,B,C,X_f),\underline{X>0} \models X_f>0 \arrow[d, "\mathrm{LU}"'] \arrow[dr, "\mathrm{LU}"] & \\
2a.\ p(5,X,A,B,C,X_f),\underline{X>0},C=1 \models X_f>0 \arrow[d, "\mathrm{LU}"'] & 2b.\ p(\Omega,X,A,B,C,X_f),\underline{X>0},C=0 \models X_f>0 \arrow[d, "\mathrm{LU}"] \\
3.\ p(\Omega,X',A,B,C,X_f),\underline{X'=X+1},\underline{X>0},C=1 \models X_f>0 \arrow[d, "\mathrm{LU}"'] & 6.\ \underline{X>0},C=0,\underline{X=X_f} \models X_f>0 \arrow[d, "\mathrm{DP}"] \\
4.\ \underline{X'=X+1},\underline{X>0},C=1,\underline{X'=X_f} \models X_f>0 \arrow[d, "\mathrm{DP}"] & 7.\ \neg\Box \\
5.\ \neg\Box &
\end{tikzcd}$$

$$\begin{tikzcd}
1.\ p(0,X,A,B,C,X_f),\underline{X>0} \models X_f>0 \arrow[d, "\mathrm{LU}"'] \arrow[dr, "\mathrm{LU}"] & \\
2a.\ p(1,X,A,B,C,X_f),\underline{X>0},A=1 \models X_f>0 \arrow[d, "\mathrm{LU}"'] & 2b.\ p(2,X,A,B,C,X_f),X>0,A\neq 1 \models X_f>0 \\
3.\ p(2,X,A,B,C,X_f),\underline{X>0},A=1 \models X_f>0 \arrow[d, "\mathrm{LU}"'] \arrow[dr, "\mathrm{LU}"] & \\
4a.\ p(3,X,A,B,C,X_f),\underline{X>0},A=1,B=1 \models X_f>0 \arrow[d, "\mathrm{LU}"'] & 4b.\ p(4,X,A,B,C,X_f),\underline{X>0},A=1,B\neq 1 \models X_f>0 \arrow[d, "\mathrm{CUT}"] \\
5.\ p(4,X,A,B,C',X_f),\underline{X>0},A=1,B=1,C'=0 \models X_f>0 \arrow[d, "\mathrm{CUT}"'] & 8.\ X_f>0,A=1,B\neq 1 \models X_f>0 \arrow[d, "\mathrm{DP}"] \\
6.\ X_f>0,A=1,B=1,C'=0 \models X_f>0 \arrow[d, "\mathrm{DP}"'] & 9.\ \neg\Box \\
7.\ \neg\Box &
\end{tikzcd}$$

Figure 6.8: Optimized Compositional Proof of Simple If Sequence Program

The procedure *prove* would then call

$$summarize(p(0,X,A,B,C,X_f),X>0 \models X_f>0).$$

which initiates a traversal of the lower proof tree in Figure 6.8 in depth-first manner. Nodes 5 and 4b of the proof tree are abstraction point, assuming that $p(4,X,A,B,C,X_f),X>0 \models X_f>0$ is not yet proved, this is then handled by Lines $\langle 9\rangle$–$\langle 11\rangle$ in Figure 6.6. In this case, the S that we choose in Line $\langle 9\rangle$ is $p(4,X,A,B,C,X_f),X>0 \models X_f>0$. This unproved assertion is added to the $NotDone^L$ set, and at the end of the execution of *summarize* is returned to *prove* to be included in *NotDone*. *Prove* then iterates over the contents of *NotDone* and prove each by again calling *summarize* (Lines $\langle 4\rangle$–$\langle 8\rangle$ in Figure 6.7). Since $p(4,X,A,B,C,X_f),X>0 \models X_f>0$ is in *NotDone*, it is therefore processed here by the call

$$summarize(p(4,X,A,B,C,X_f),X>0 \models X_f>0)$$

which produces the traversal of the upper tree in Figure 6.8. This traversal produces no more assertion in *NotDone*, and hence after the return from *summarize* the program terminates.

We note that here we obtain a proof which is smaller than both Figure 5.14 (Section 5.8.4) and Figure 6.5 (Section 6.2.1). Due to compositionality, it is not necessary to repeat the proof of the upper tree at 5 and 4b of the lower tree, and due to summarization, 2b of the lower tree becomes redundant to 3.

Chapter 7

Toward a Basic Algorithm for Recursive Assertions

Here we discuss the verification of two kinds of recursive assertions we have previously mentioned: relative safety assertions and general recursive assertions we encounter in array and pointer data structure verification. The proof of relative safety assertions we have discussed so far is automatable, and we provide the basic algorithm in Section 7.1. We also provide discussions on how general recursive assertions in data structure verification can be automated in Section 7.2.

7.1 Algorithm for Proving Relative Safety

We devise an algorithm for proving relative safety in Figure 7.1. It is based on our straightforward algorithm in Figure 6.1. Similar to Figure 6.1, it uses the set of assumed assertions only instead of global tabling, but different from it, it allows for application of CUT rule in order to check whether an assertion is related by a relative safety property (e.g., symmetric) to another in the set of assumed assertions.

Since the proof of a relative safety assertion can employ other relative safety assertions that are proved separately, we allow the invocation of the procedure *prove* with a nonempty assumed assertions $\tilde{A}$ at $\langle 1 \rangle$. In addition, the proof of a relative safety assertion using our proof method requires the application of RU rule. The checking of $G'' \models H$ at $\langle 2 \rangle$ would require a search process that uses only the RU and DP rules. $G'' \models H$ is established when after a number of applications of RU we are able to conclude the final obligation using DP. The right unfold via RU here can

```
        program
⟨1⟩       $prove(\tilde{A}, G \models H)$
        end program

        proc $prove(\tilde{A}, G \models H)$
          for each $(G'' \in \{G' | G \models G'\})$ do
⟨2⟩         if $(G'' \models H$ is provable, or
                there is $A \in \tilde{A}$
⟨3⟩             such that $A \rhd (G'' \models H))$ then
              return $Success$
            end if
          end for
          $\tilde{A} := \tilde{A} \cup \{G \models H\}$
          $F := unfold(G)$
          if $(F \neq \emptyset)$ then
            for each $(g \in F)$ do
              $prove(\tilde{A}, g \models H)$
            end for
            return $Success$
          end if
          abort
        end proc
```

Figure 7.1: Relative Safety Prover Algorithm

be done either by depth-first or breadth-first strategy. It is often useful to give a bound on the depth of the right unfolding since it is possible that further right unfolding beyond certain depth is futile. The proof process has to progress by more steps of left unfold (LU) when the right unfolding has reached its depth bound without finding a proof.

7.2 Toward Automation of Data Structure Proof

Our main compositional algorithm of Figure 6.7 for program analysis and verification can also be used for verifying programs with pointer data structures. However, in this case, the proof is not completely automatic. For explanation, we provide a simpler algorithm for proving data structure property in Figure 7.2. It is similar to the algorithm in Figure 6.4 for intermittent abstraction and global tabling, and in fact Figure 6.4 is also a correct algorithm for data structure verification, but in Figure 7.2 we opt to simply use the set of assumed assertions, which is enough for our purpose instead of global table. We note that our sample data structure proofs in Chapter 5 and appendix Section B.1 have been done without the redundancy checks made possible by global tabling.

Figure 7.2 is not completely automatic because at the moment the operations at ⟨1⟩, ⟨2⟩,

```
        program
           $prove(\emptyset, G \models H)$
        end program

        proc $prove(\tilde{A}, G \models H)$
⟨1⟩        if $(G \models H$ is provable$)$ then
              return $Success$
           end if
⟨2⟩        if $(abstraction\_point(G)$ and $G \models G')$ then
              $G := G'$
           end if
           if (There is $A \in \tilde{A}$
⟨3⟩              such that $A \rhd (G \models H))$ then
              return $Success$
           end if
           $\tilde{A} := \tilde{A} \cup \{G \models H\}$
           $F := unfold(G)$
           if $(F \neq \emptyset)$ then
              for each $(g \in F)$ do
                 $prove(\tilde{A}, g \models H)$
              end for
              return $Success$
           end if
           abort
        end proc
```

Figure 7.2: Simple Algorithm for Proving Data Structure Property

and ⟨3⟩ are not fully automatic. The test at each of these points is a proof of general recursive assertion with constraints on arrays, which often requires right unfolding (RU rule) due to the atoms in the rhs of the assertion, or seemingly arbitrary generalization via CUT rule. To be more specific, suppose that we want to prove $p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \phi \models q(\tilde{Y}), \psi$. Here, often we need to generalize the lhs, such as by removing some literals. The purpose of this is so that we may conclude the proof later by the application of coinduction (AP).

Fortunately, in the domain of recursive data structure verification, the proof steps of general recursive assertions are not completely arbitrary. Here we provide a strategy that would work for our examples.

We assume that a general recursive assertion has the form

$$p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \phi \models q(\tilde{Y}), \psi,$$

where ϕ and ψ are constraints. This form with only one atom at the rhs is general enough since

$G \models q_1(\ldots), q_2(\ldots), \ldots, q_n(...)$, can be proved by separately proving $G \models q_1(\ldots)$, $G \models q_2(\ldots)$, $\ldots$, and $G \models q_n(\ldots)$. We also assume that no $p_i(\tilde{X}_i)$, for $1 \leq i \leq m$ is a predicate of program model (the predicate p we have been using so far). The above general recursive assertion can be encountered as a subsumption test for applying AP, the residual obligation after an application of AP, or after applying LU rule using a constraint fact.

Given the above general recursive assertion, we perform the following steps:

1. First we try to prove the assertion directly by application of DP, or by a number of applications of RU, which is followed by a single DP. If this prove succeeds, we return to the caller, reporting a success. Here we may also utilize separation principle (SEP) and/or array index principle (AIP) to simplify the assertions. We often also need to discover a substitution to existential variables of the rhs of the assertion to allow the application of DP. It is straightforward to implement an incomplete automated procedure for this purpose which is based on unification.

2. We try to apply AP if there is an ancestor of this assertion in the proof tree with the same multiset of predicates in the lhs goal. Any application of AP involves the following two:

   (a) Subsumption test.

   (b) Proof of the residual assertion.

   Both are proved independently, and each is also a proof of general recursive assertion. If we succeed with both proofs, we signal a success to the caller.

3. If with the above cases we are unable to establish a proof, we attempt to perform more left unfold (LU). In the recursive data structure verification, we can impose some restriction on the way we perform the unfolds.

   Since we are dealing with predicates defining recursive data structures which are typically simple (trees, lists, etc.), in most cases, each atom needs only one level of unfold. Therefore, an atom p_i, where $1 \leq i \leq n$ which is a result of previous left unfold can be given lower priority to be unfolded at the current stage of the proof. This results in fair atom selection in the unfold. That is, if the proof does not conclude earlier, all atoms in the initial obligation will eventually be unfolded.

   Here there are two cases:

(a) In case no more unfold possible, signal failure to the caller, and terminate the whole proof process.

(b) Otherwise, try to unfold the goal, and for each child goal, repeat from step 1.

Chapter 8

Implementation and Experiments

8.1 Basic Implementation in CLP($\mathcal{R}$)

In this section we discuss the implementations of the algorithms given in Chapters 6 and 7 in the CLP($\mathcal{R}$) programming language [110, 94].

8.1.1 Verification Run with SLD Resolution

Let us start with an implementation of the CLP program that we want to reason about, whose general form is Program 8.1. From the discussion in Section 2.5.2, we know that we can already perform some reasoning on the program using SLD resolution. For example, proving $p(\tilde{X}), \varphi(\tilde{X}) \models \Box$ is the same as posing the query clause

$$\Box \texttt{ :- } p(\tilde{X}), \varphi(\tilde{X}).$$

For example, we may pose the query $\Box \texttt{ :- } p(2,2,X,Y)$. to Program 3.15 (Page 62) in order to establish $p(2,2,X,Y) \models \Box$.

In this case standard CLP resolution already implements LU and DP proof rules, because one step of resolution corresponds to generation of resolvents using all program rules. DP here is manifested by the checking of the satisfiability of a conjunction (sequence) of constraints. In case we encounter a goal clause $\Box \texttt{ :- } \varphi$, where the goal φ is just a satisfiable sequence of constraints, a reasonable CLP system would automatically report a refutation.

Example 8.1. For example, let us again refer to the proof of $p(2,2,X,Y) \models \Box$ (Figure 5.9) in Section 5.6. First note that in one step the resolvents of the query $\Box \texttt{ :- } p(2,2,X,Y)$ (name this

$$
\begin{array}{rcl}
p(\tilde{X}) & \texttt{:-} & \alpha_1(\tilde{X},\tilde{X}'),p(\tilde{X}'). \\
 & \vdots & \\
p(\tilde{X}) & \texttt{:-} & \alpha_n(\tilde{X},\tilde{X}'),p(\tilde{X}'). \\
p(\tilde{X}) & \texttt{:-} & \beta_1(\tilde{X}). \\
 & \vdots & \\
p(\tilde{X}) & \texttt{:-} & \beta_m(\tilde{X}).
\end{array}
$$

Program 8.1: First Engine

$$
\begin{array}{r|ll}
1 & p(X),X=2Y+1 \models \Box & \\
\hline
2a & 0=2Y+1 \models \Box & \textsc{lu}\ 1 \\
2b & p(X-2),X=2Y+1 \models \Box & \textsc{lu}\ 1 \\
3 & \neg\Box & \textsc{dp}\ 2a \\
4 & \neg\Box & \textsc{ap}\ 1,2b
\end{array}
$$

$$
\begin{array}{r|ll}
4s.1 & p(X-2),X=2Y+1 & \\
 & \quad \models p(X-2),X-2=2\times ?Z+1 & \textsc{ap}\ 1,2b \\
\hline
4s.2 & \neg\Box & \textsc{dp}\ 4s.1
\end{array}
$$

$$
\begin{array}{r|ll}
4r.1 & \Box,X=2Y+1 \models \Box & \textsc{ap}\ 1,2b \\
\hline
4r.2 & \neg\Box & \textsc{dp}\ 4r.1
\end{array}
$$

Figure 8.1: Proof of $p(X),X=2Y+1 \models \Box$

clause κ_8) produced using the clauses of Program 3.15 are

$$\Box \;\texttt{:-}\; p(1,2,X,Y),(Y=0 \vee X<Y). \quad \kappa_9 = \mathit{resolv}_{p(2,2,X,Y)}(\kappa_8,\kappa_3)$$

$$\Box \;\texttt{:-}\; p(2,1,X,Y),(X=0 \vee Y<X). \quad \kappa_{10} = \mathit{resolv}_{p(2,2,X,Y)}(\kappa_8,\kappa_6)$$

Obviously κ_9 is the obligation 2a in Figure 5.9, while κ_{10} is the obligation 2b in Figure 5.9.

Example 8.2. As another example, let us prove the assertion $p(X),X=2Y+1 \models \Box$ on Program 5.1 (Section 5.1). The proof is shown in Figure 8.1. Given Program 5.1, we pose the query $\Box \;\texttt{:-}\; p(X),X=2Y+1$ (call this clause κ_3), which has the following two resolvents:

$$\Box \;\texttt{:-}\; 0=2Y+1. \qquad \kappa_4 = \mathit{resolv}_{p(X)}(\kappa_3,\kappa_1)$$

$$\Box \;\texttt{:-}\; p(X-2),X=2Y+1. \quad \kappa_5 = \mathit{resolv}_{p(X)}(\kappa_3,\kappa_2)$$

Here, κ_4 is the obligation 2a in Figure 8.1, and κ_5 is the obligation 2b in Figure 8.1. Assuming the existence of an automated integer constraint solver, we can simplify the expression $0=2Y+1$ in κ_4 into $\Box$ hence producing $\Box \;\texttt{:-}\; \Box$. This corresponds to a direct proof (DP) step producing the obligation 3 in Figure 8.1.

8.1.2 Checking Assertion Entailment

To implement AP or use the redundancy principle, we implement the global tabling mentioned in Section 6.1 (Figure 6.2). Here we need to test for assertion entailment, which we perform in the following way. Assume two assertions $G \models \square$ and $H \models \square$, where $H \models \square$ is stored in the table. We consider the goal G to be subsumed by H when G and H are renamed apart, the goal G is $p(\tilde{X}), G'$, the goal H is $p(\tilde{Y}), H'$, and $G' \models H', \tilde{X} = \tilde{Y}$. In our implementation, which will be detailed later, the subsumed and subsuming goals (G and H respectively) are always renamed apart. Now, $G' \models H', \tilde{X} = \tilde{Y}$ is equivalent to

$$\langle \exists var(G') - var(\tilde{X}) : G' \rangle \Rightarrow \langle \exists var(H') - var(\tilde{Y}) : H' \rangle, \tilde{X} = \tilde{Y}. \tag{8.1}$$

The expression $\langle \exists var(G') - \tilde{X} : G' \rangle$ is a *projection* of G' into the variables $\tilde{X}$. Similarly, $\langle \exists var(H') - \tilde{Y} : H' \rangle$ is a projection of H' into the variables $\tilde{Y}$. Such projections are computed automatically and efficiently by CLP($\mathcal{R}$). The residual obligation here is $\square \models \square$ which is trivial.

Example 8.3. For example, in the mutual exclusion proof of the two-process bakery algorithm (Figure 5.9), we already assume the existence of a constraint solver capable of computing projections. Now let us compute κ_{11} from κ_9 as follows:

$$\square \ \text{:-}\ \ p(0,2,X',Y), (Y = 0 \vee X < Y), X = Y + 1. \quad \kappa_{11} = resolv_{p(1,2,X,Y)}(\kappa_9, \kappa_2)$$

Since the variable X no longer appears in the arguments of the atom $p(0,2,X',Y)$, CLP($\mathcal{R}$) automatically projects it out and instead process the much simple clause

$$\square \ \text{:-}\ \ p(0,2,X',Y), Y = 0. \quad \text{Simplify } \kappa_{11} \text{ with constraint solving.}$$

This clause corresponds to the obligation 3a in Figure 5.9.

8.1.3 Storing in Global Table

To implement the global tabling mechanism, we need a way to store assertions persistently. Notice again that to establish subsumption we test whether a goal is subsumed by a goal already

```
store(X̃) :- dump(X̃, X̃′, S), negate_all(S, S′), assert(t(X̃′, S′)).

negate_all([], []).
negate_all([C|R], [C′|R′]) :- negate(C, C′), negate_all(R, R′).
```

Program 8.2: *Store*

stored in the table (Formula (8.1)), which is equivalent to the unsatisfiability of

$$\langle \exists var(G') - var(\tilde{X}) : G' \rangle \wedge \langle \exists var(H') - var(\tilde{Y}) : \neg H' \rangle, \tilde{X} = \tilde{Y}. \tag{8.2}$$

The subsumption test that we implement is actually the unsatisfiability test of (8.2). For efficiency, it is therefore desirable not to simply to store an assertion, but to store the negation of the constraints part $\neg H'$ of its lhs such that $\neg H'$ need not be recomputed whenever we test for subsumption.

For negating and storing a goal, we implement the *store* procedure shown as Program 8.2. Our *store* procedure uses CLP($\mathcal{R}$)'s built-in *dump* procedure to extract the constraints on the variables $\tilde{X}$ into a list of syntactic constraints S on the variables $\tilde{X}'$. We then compute the syntactic negation by the *negate_all* procedure, and then store the negation in the persistent store using CLP($\mathcal{R}$)'s *assert* built-in procedure. Note that in (8.2) H' can be a sequence (conjunction) of constraints, say $\varphi_1, \ldots, \varphi_n$. *Negate_all* computes the list $[\varphi'_1, \ldots, \varphi'_n]$, where φ'_i is a negation of φ_i, for $1 \leq i \leq n$. The meaning of this list is the disjunction $\bigvee_{i=1}^{n} \varphi'_i$.

8.1.4 Algorithm with Table Checking and Storing

It is best to combine the table checking and storing into a single procedure, as shown in Program 8.3. In the *check_and_store* procedure, we first test whether a set of constraints is not subsumed by the table. If this is the case, we then execute the *store* procedure to store it in the table. The actual subsumption test is implemented in the *subsumed* procedure by testing whether there is an item $t(\tilde{X}, S)$ in the table, where S is a negated constraints list with $\tilde{X}$ as the variables, and where all constraints in S is unsatisfiable. Recall that the meaning of S is a disjunction of its elements. Hence, the unsatisfiability of S requires unsatisfiability of each of its elements. *All_unsat* iterates through the elements of S and checks for the unsatisfiability of each. The *satisfiable* procedure, whose code is not shown here, is a syntactic constraint evaluator.

Our remaining task is to put *check_and_store* in the appropriate places in Program 8.1. We

$$
\begin{array}{l}
\mathit{check\_and\_store}(\tilde{X}) \ \texttt{:-}\ \mathit{not\ subsumed}(\tilde{X}), \mathit{store}(\tilde{X}).\\[1em]
\mathit{subsumed}(\tilde{X}) \ \texttt{:-}\ t(\tilde{X}, S), \mathit{all\_unsat}(S).\\[1em]
\mathit{all\_unsat}([\,]).\\
\mathit{all\_unsat}([C|R]) \ \texttt{:-}\ \mathit{not\ satisfiable}(C), \mathit{all\_unsat}(R).
\end{array}
$$

Program 8.3: *Check_and_Store*

$$
\begin{array}{lll}
p(\tilde{X}) & \texttt{:-} & \mathit{check\_and\_store}(\tilde{X}), q(\tilde{X}).\\[1em]
q(\tilde{X}) & \texttt{:-} & \alpha_1(\tilde{X}, \tilde{X}'), p(\tilde{X}').\\
 & \vdots & \\
q(\tilde{X}) & \texttt{:-} & \alpha_n(\tilde{X}, \tilde{X}'), p(\tilde{X}').\\
q(\tilde{X}) & \texttt{:-} & \beta_1(\tilde{X}).\\
 & \vdots & \\
q(\tilde{X}) & \texttt{:-} & \beta_m(\tilde{X}).
\end{array}
$$

Program 8.4: Second Engine

refer again to our algorithm in Figure 6.2, where table checking is the first routine executed in the procedure *prove*, which is then followed by the storing. Accordingly, *p* must first call *check_and_store* before executing the unfolds. In Program 8.4 we therefore define a new procedure *q*, which actually executes the unfolds and recursively calls *p*. We then modify *p* to first call *check_and_store* before calling *q* to execute further unfolds.

Program 8.4 is already a complete implementation of the algorithm in Figure 6.2 for proving $G \models \Box$. In order to separate the verification machinery from the problem so that we can use the same engine for separate problems, we separate the problem-dependent constraints $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ into predicates *trans* and *init* in Program 8.5.

8.2 Specialization to Programs

An important application of our proof method is for reasoning about programs and timed automata. As we have discussed in Chapter 3, programs have program points, similarly timed automata have location identifiers which are nonnegative integers. We usually assign program point variables to the leftmost *n* arguments of *p*, where *n* is the number of concurrent processes or automata.

When running our prover engine on program or timed automata problems, all goals always

```
$trans(\tilde{X},\tilde{X}')$ :- $\alpha_1(\tilde{X},\tilde{X}')$.
                  ⋮
$trans(\tilde{X},\tilde{X}')$ :- $\alpha_n(\tilde{X},\tilde{X}')$.

$init(\tilde{X})$ :- $\beta_1(\tilde{X})$.
          ⋮
$init(\tilde{X})$ :- $\beta_m(\tilde{X})$.

$p(\tilde{X})$ :- $check\_and\_store(\tilde{X}), q(\tilde{X})$.

$q(\tilde{X})$ :- $trans(\tilde{X},\tilde{X}'), p(\tilde{X}')$.
$q(\tilde{X})$ :- $init(\tilde{X})$.
```

Program 8.5: Third Engine

have the program point arguments *ground*, that is, they have known integer values. Therefore in the subsumption check logically formalized by (8.2), we need not have constraints on program points or timed automata locations in H' since we would only test for subsumption of goals with the same program point values. This improves, or will potentially improve, the efficiency of our implementation for the following reasons:

- We can avoid negating the groundings of program points or locations, which would improve the efficiency of our algorithm.
- CLP($\mathcal{R}$) indexes clauses based on the numeric value of its first (leftmost) argument. Therefore, having the first argument of the table clauses t in Program 8.2 to be numeric is important for efficiency of table checking.
- We could potentially implement more efficient problem-specific indexing mechanisms.

To accommodate the new table storing and subsumption checking, we modify the predicates we have introduced before, by separating the program points from the rest of the arguments. We show the modified predicates as Program 8.6. In the predicates *trans* and *init*, we now separate the first ground values $\tilde{r}_1, \ldots, \tilde{r}_n, \tilde{r}'_1, \ldots, \tilde{r}'_n, \tilde{s}_1, \ldots, \tilde{s}_m$. In the other predicates we separate the arguments into $\tilde{L}$ and $\tilde{X}$. The negation of constraints in *store* only involves constraints on the arguments $\tilde{X}$, and not $\tilde{L}$. In *subsumed* we simply match the ground values of $\tilde{L}$ with which the procedure is called with the table without testing for unsatisfiability (of say, $L = 1 \wedge L \neq 1$).

$$
\begin{array}{l}
\mathit{trans}(\tilde{r}_1,\tilde{X},\tilde{r}'_1,\tilde{X}') \ \texttt{:-}\ \alpha_1(\tilde{X},\tilde{X}').\\
\qquad\qquad\vdots\\
\mathit{trans}(\tilde{r}_n,\tilde{X},\tilde{r}'_n,\tilde{X}') \ \texttt{:-}\ \alpha_n(\tilde{X},\tilde{X}').\\
\\
\mathit{init}(\tilde{s}_1,\tilde{X}) \ \texttt{:-}\ \beta_1(\tilde{X}).\\
\qquad\vdots\\
\mathit{init}(\tilde{s}_m,\tilde{X}) \ \texttt{:-}\ \beta_m(\tilde{X}).\\
\\
\mathit{check\_and\_store}(\tilde{L},\tilde{X}) \ \texttt{:-}\ \mathit{not\ subsumed}(\tilde{L},\tilde{X}),\mathit{store}(\tilde{L},\tilde{X}).\\
\\
\mathit{subsumed}(\tilde{L},\tilde{X}) \ \texttt{:-}\ t(\tilde{L},\tilde{X},S),\mathit{all\_unsat}(S).\\
\\
\mathit{store}(\tilde{L},\tilde{X}) \ \texttt{:-}\ \mathit{dump}(\tilde{X},\tilde{X}',S),\mathit{negate\_all}(S,S'),\mathit{assert}(t(\tilde{L},\tilde{X}',S')).\\
\\
p(\tilde{L},\tilde{X}) \ \texttt{:-}\ \mathit{check\_and\_store}(\tilde{L},\tilde{X}),q(\tilde{L},\tilde{X}).\\
\\
q(\tilde{L},\tilde{X}) \ \texttt{:-}\ \mathit{trans}(\tilde{L},\tilde{X},\tilde{L}',\tilde{X}'),p(\tilde{L}',\tilde{X}').\\
q(\tilde{L},\tilde{X}) \ \texttt{:-}\ \mathit{init}(\tilde{L},\tilde{X}).
\end{array}
$$

Program 8.6: Fourth Engine

8.3 Handling Program Data Types

The domains of variables in CLP($\mathcal{R}$) are only functors and real numbers. In this section we discuss various possible variants of implementing *store* and *subsumed* depending on the intended data type of the verification problem at hand.

8.3.1 Tabling Integer in CLP($\mathcal{R}$)

As we have mentioned in Section 5.10.1, there is an inherent incompleteness when verifying integer problems due to our use of CLP($\mathcal{R}$). However, there is a technique that we can employ to increase precision. The solution here is that in the implementation of *negate* (Program 8.2) we should never negate constraints on integer variables to strict inequality.

All said, other than not affecting the soundness of verification result, we also believe the problem considered here has a rare occurrences. Therefore this technique is not yet implemented in our actual prover prototype.

Example 8.4. Suppose that during the run of the prover we obtain earlier the goal $p(15,X),X \leq 5 \models \square$, which we store in the table (using *assert*) after negating $X \geq 5$ to $X < 5$, as the constraint fact

$$t(15, [X > 5]).$$

A problem occurs when X is intended to be an integer variable: Suppose that in another part of the tree we prove the assertion

$$p(15, X), X < 6 \models \Box.$$

Since X is an integer variable, the lhs goal should be subsumed by the lhs goal of $p(15, X), X \leq 5 \models \Box$.

The prover engine checks whether the last assertion has been tabled or not by executing the call $\mathit{subsumed}(15, X)$ from within *check_and_store*. This leads to the execution of the goal $X < 6, \mathit{all\_unsat}([X > 5])$. In real domain $X < 6 \land X > 5$ has solutions, therefore *all_unsat* and *subsumed* fail, and the assertion $p(15, X), X < 6 \models \Box$ is wrongly considered by our implementation to be not subsumed by CLP($\mathcal{R}$).

The solution here is that instead of storing $t(15, [X > 5])$ in the table, we should have stored $t(15, [X \geq 6])$, that is, we do not negate the constraint to strict inequality.

8.3.2 Subsumption of Functors in CLP($\mathcal{R}$)

We need to handle functors to verify problems such as the statechart example in Section 3.7. For this, we need to modify how the table is constructed and used in subsumption check. An obvious way to store a term in the table is by storing it as is, as shown in Program 8.7 (no "negating" as in Program 8.2).

For the subsumption check, note that a term T is subsumed by another term S when there is a substitution σ of the variables of S such that $S\sigma = T$. Our task is therefore to generate one such substitution. This is implemented in our new version of *subsumed* in Program 8.7. *Subsumed* calls *all_subsumed*, which further calls *subsumed_aux*. *Subsumed_aux* uses CLP($\mathcal{R}$)'s *var* and $=..$ built-in predicates. *Var* is used to test whether a variable has a constant valuation or not, while $=..$ is used to decompose a term into a list containing its head and arguments.

8.3.3 Tabling Finite Domain Data in CLP($\mathcal{R}$)

The main use of this type is to be able to handle finite integer constraints such as $K \neq \alpha$ in the transition definitions, where K is of finite subset of integers and α a constant in the domain of K without translating it into the disjunction $K < \alpha \lor K > \alpha$. The problem with disjunction is

```
store(X̃) :- assert(t(X̃)).

subsumed(X̃) :- t(Ỹ), subsumed_all(X̃, Ỹ).

all_subsumed([], []).
all_subsumed([X|R], [Y|S]) :-
      subsumed_aux(X, Y), all_subsumed(R, S).

subsumed_aux(A, B) :- var(B), !, A = B.
subsumed_aux(A, B) :- not var(A),
      A = ..[H|R], B = ..[H|S],
      all_subsumed(R, S).
```

Program 8.7: *Store* and *Subsumed* for Handling Terms

that it results in more branchings in the proof tree, which is against our effort to keep the proof tree as small as possible. For example, in the Fischer's mutual exclusion algorithm example in Section 4.5.2, we have an if conditional which condition is an inequality. Since CLP($\mathcal{R}$) has no representation for inequality constraint, the actual CLP($\mathcal{R}$) implementation of the sixth clause in Program 4.20 are the two clauses

$$p(0,L_2,T_1+1,T_2,K) \;\texttt{:-}\; p(3,L_2,T_1,T_2,K), T_1 \leq T_2, K < 1.$$
$$p(0,L_2,T_1+1,T_2,K) \;\texttt{:-}\; p(3,L_2,T_1,T_2,K), T_1 \leq T_2, K > 1.$$

These two rules would result in branching in the proof tree, possibly enlarging its size. Similar case can be found in Szymanski's algorithm (also in Section 4.5.2), where there are if conditionals with equality condition. The failure ("else") path of the conditional therefore has inequality as its guard.

To solve this problem we introduce special terms of the form $room(List)$, which we call *room terms*[1], where *List* is a list of 0, 1, or "any" value, which in CLP($\mathcal{R}$) is denoted by _[2], with the convention that only one element with value 1 is allowed, and in case when there exists a 1, no _ is allowed in the list, and the list cannot be all 0. Therefore terms such as $room([0,1,0])$ and $room([\_,0,0])$ are well formed, while terms such as $room([0,0,0])$ or $room([1,1,\_])$ are not.

Each element of *List* represents an element of a finite domain. For example, in the term $room([a_1,a_2,a_3])$, a_1, a_2, and a_3 may correspond to the colors "red," "green," "blue," or the numbers "5," "3," "4," respectively, depending on the interpretation given by the user. Suppose

[1]The naming is inspired by the flag variables x_1, x_2 in two-process Szymanski's mutual exclusion algorithm which point to some finite number of "waiting rooms."

[2]_ can be understood to be some fresh, otherwise unconstrained variable.

that we adopt the numeric interpretation. The term *room*$([0,1,0])$ now represents exactly the number “3.” For this to be so, the element of the list which correspond the object “3” is set to 1 while the rest of the elements are set to 0.

We use our term not only to represent a value, but also a set of values in the finite domain. To represent a set of values that does not include “3,” we need a term which abstracts *room*$([1,0,0])$ (representing “5”) and *room*$([0,0,1])$ (representing “4”). The term is *room*$([\_,0,\_])$, denoting “not 3.” Note that now the middle element of the list which corresponds to “3” is 0, while the rest are given _ (“any” value). Here we do not need to use disjunction to represent an inequality.

We now explain how to store and use room terms for subsumption checks efficiently. For this purpose, similar to numeric variables, we also pre-process a room term by “negating” it before storing, such that the subsumption check can be done simply by unification (defined on Page 2.5.2).

We first define an asymmetric *negation* mapping of list elements: $\{0 \mapsto \_, 1 \mapsto 0, \_ \mapsto 0\}$, which implementation for negating room terms is shown as Program 8.8. When the call *room_negate*$(A,\, B)$ succeeds given room term A as input, we call B to be the room negation of A (but not the vice versa due to the asymmetry of the mapping). In addition to mapping list elements, *room_negate* also adds the constraint that the sum of all non-ground elements in the list of room term B is 1. For example, the term *room*$([X,0,Y]), X+Y=1$, where X and Y are non-ground (“any”), is the room negation of *room*$([0,1,0])$ (but again, not the vice versa).

Other predicates defined in Program 8.8 include *room_negate_all* which construct the negations of room terms in a list, and *none_unifiable* to test of non-unifiability of corresponding room terms of two lists.

We now specify a subset of the program variables with finite domains, and we accommodate these into a new version of our prover engine, shown as Program 8.9. In Program 8.9 we denote a list of room term variables by $\tilde{F}$, $\tilde{F}'$, or $\tilde{G}$.

We need the following result:

Proposition 8.1. Correctness of Room Term Subsumption. The term *room*$(List_1)$ is not unifiable with *room*$(List_2)$, where *room*$(List_2)$ is the room negation of *room*$(List_3)$, if and only if the set of values represented by *room*$(List_1)$ is included in the set of values represented by *room*$(List_3)$.

```
room_negate(room(L), room(L')) :-
    room_negate_aux(L, L', 1).

room_negate_aux([], [], 0).
room_negate_aux([A|R], [0|S], X) :- var(A), !, room_negate_aux(R, S, X).
room_negate_aux([0|R], [A|S], A+X) :- room_negate_aux(R, S, X).
room_negate_aux([1|R], [0|S], X) :- room_negate_aux(R, S, X).

room_negate_all([], []).
room_negate_all([X|R], [Y|S]) :- room_negate(X, Y), room_negate_all(R, S).

none_unifiable([], []).
none_unifiable([X|R], [Y|S]) :- not X = Y, none_unifiable(R, S).
```

Program 8.8: *Room_Negate*, *Room_Negate_All*, and *None_Unifiable*

Proof. First we prove the only if case. Note that since $room(List_2)$ is a room negation of $room(List_3)$, $List_2$ does not contain a 1. Hence, the term $room(List_1)$ is not unifiable with $room(List_2)$ if and only if $List_1$ has a 1 at position i, while $List_2$ has a 0 at the same position. The 0 at position i of $List_2$ can only be a result of room-negating either _ or 1. In either case, the set of values represented by $room(List_3)$ necessarily include those represented by $room(List_1)$.

Next we prove the if case by contraposition that if $room(List_1)$ is unifiable with $room(List_2)$, then necessarily the set of objects represented by $room(List_1)$ is not included in the set of objects represented by $room(List_3)$. Note that $List_2$ contains only 0s and some $k > 1$ non-grounds, $g_1, \ldots, g_k$, such that $g_1 + \ldots + g_k = 1$.

The only case when the above is violated is when given the m.g.u. (defined on Page 34) μ of $room(List_1)$ and $room(List_2)$ is such that $room(List_1)\mu = room(List_2)\mu = room([0, \ldots, 0])$. To see this, first we suppose that the result of unification contains a _. Then necessarily both $List_1$ and $List_2$ contain _ at the same position. Since _ is a room negation of 0, this means $room(List_1)$ represents some objects not represented by $room(List_3)$. Next, suppose that the result of unification contains a 1. Then necessarily $List_1$ contains a 1 at that position while $List_2$ contains a _ at the same position. Since _ is a negation of 0, then the object represented by $room(List_1)$ cannot be included in the set of objects represented by $room(List_3)$.

However, the unification with the result $room(List_1) = room(List_2) = room([0, \ldots, 0])$ is not possible since $k > 1$ and $g_1 + \ldots + g_k = 1$. □

Using the above theorem, it is thus possible to check whether the set of values represented by $room(List_1)$ encountered during search, is subsumed by the set of values represented by

```
trans(r̃_1, X̃, F̃, r̃'_1, X̃', F̃') :- α_1(X̃, F̃, X̃', F̃').
                  ⋮
trans(r̃_n, X̃, F̃, r̃'_n, X̃', F̃') :- α_n(X̃, F̃, X̃', F̃').

init(s̃_1, X̃, F̃) :- β_1(X̃, F̃).
          ⋮
init(s̃_m, X̃, F̃) :- β_m(X̃, F̃).

check_and_store(L̃, X̃, F̃) :- not subsumed(L̃, X̃, F̃), store(L̃, X̃, F̃).

subsumed(L̃, X̃, F̃) :- t(L̃, X̃, G̃, S), none_unifiable(F̃, G̃), all_unsat(S).

store(L̃, X̃, F̃) :- dump(X̃, X̃', S), negate_all(S, S'),
      room_negate_all(F̃, F̃'), assert(t(L̃, X̃', F̃', S')).

p(L̃, X̃, F̃) :- check_and_store(L̃, X̃, F̃), q(L̃, X̃, F̃).

q(L̃, X̃, F̃) :- trans(L̃, X̃, F̃, L̃', X̃', F̃'), p(L̃', X̃', F̃').
q(L̃, X̃, F̃) :- init(L̃, X̃, F̃).
```

Program 8.9: Fifth Engine

$room(List_3)$ encountered earlier by checking non-unifiability of $room(List_1)$ to $room(List_2)$.

Sometimes in the proof tree some ill-formed room terms can be constructed. An example of the problem is when $room([0,0,0])$[3] is generated due to unification of $room([0,\_,\_])$ with $room([\_,0,\_])$, and then with $room([\_,\_,0])$ in a single search path. Either in the position of $room(List_1)$ (the term to be compared with a previously encountered one), or $room(List_3)$ (the term previously encountered to be compared with) mentioned in Proposition 8.1, the subsumption check always detects unifiability. This problem can be solved using the mechanism for axiom checking to be introduced in Section 8.6, where for all variables of room term data type, we add the requirement that the occurrence of 1s is exactly one.

8.4 Implementing Intermittent Abstraction

Here we discuss an implementation of the algorithm of Figure 6.4 for program analysis and verification, specialized to predicate abstraction. In Figure 6.4, before testing for subsumption we test whether the constraints on the variables $\tilde{L}$ and $\tilde{X}$ is an abstraction point (the condition $abstraction\_point(G)$). In this case, we replace $\tilde{X}$ with new variables having weaker constraints.

[3]When we must provide an intuitive interpretation, this has a ’no value’ interpretation.

```
abstract(l̃, X̃, X̃′)  :-  once(abstract_1(X̃, X̃′)).
abstract(L̃, X̃, X̃).

abstract_1(X̃, X̃′)  :-  not ¬φ_1(X̃), φ_1(X̃′), abstract_2(X̃, X̃′).
abstract_1(X̃, X̃′)  :-  not φ_1(X̃), ¬φ_1(X̃′), abstract_2(X̃, X̃′).
abstract_1(X̃, X̃′)  :-  abstract_2(X̃, X̃′).
          ⋮
abstract_{k−1}(X̃, X̃′)  :-  not ¬φ_{k−1}(X̃), φ_{k−1}(X̃′), abstract_k(X̃, X̃′).
abstract_{k−1}(X̃, X̃′)  :-  not φ_{k−1}(X̃), ¬φ_{k−1}(X̃′), abstract_k(X̃, X̃′).
abstract_{k−1}(X̃, X̃′)  :-  abstract_k(X̃, X̃′).

abstract_k(X̃, X̃′)  :-  not ¬φ_k(X̃), φ_k(X̃′).
abstract_k(X̃, X̃′)  :-  not φ_k(X̃), ¬φ_k(X̃′).
abstract_k(X̃, X̃′).
```

Program 8.10: *Abstract* and *Abstract*$_1$ to *Abstract*$_k$

```
p(L̃, X̃, F̃, )  :-  abstract(L̃, X̃, X̃′), check_and_store(L̃, X̃′, F̃), q(L̃, X̃′, F̃).

q(L̃, X̃, F̃)  :-  trans(L̃, X̃, F̃, L̃′, X̃′, F̃′), p(L̃′, X̃′, F̃′).
q(L̃, X̃, F̃)  :-  init(L̃, X̃, F̃).
```

Program 8.11: Sixth Engine

In our implementation, an abstraction point is determined from the values of program points or automata locations alone. We show our abstraction routine as Program 8.10, where program point values are denoted using the first argument $\tilde{l}$ of *abstract*. When *abstract* is called, the program point values must match $\tilde{l}$. We use *abstract* only for abstracting numeric constraints, and not finite-domain constraints, since abstraction of finite-domain constraints are not required in our experiments.

Abstract calls *abstract*$_1$, *abstract*$_1$ in turn calls *abstract*$_2$, where constraints are actually abstracted. Recall that in Section 5.8.2, we are given a set of predicates $\varphi_1, \ldots, \varphi_k$. Then we abstract a set of constraints ϕ to a conjunction $\varphi'_1, \ldots, \varphi'_k$, where φ'_i is φ_i when we can decide that $\phi \Rightarrow \varphi_i$, $\neg\varphi_i$ when we can decide $\phi \Rightarrow \neg\varphi_i$, or $\neg\Box$ when we can decide neither. We straightforwardly implement this as the procedures *abstract*$_i$ (for $1 \leq i \leq k$) in Program 8.10.

Program 8.11 updates the p procedure in Program 8.9 by adding a call to *abstract*.

```
permute(L̃,X̃,F̃,L̃′,X̃′,F̃′)  :-  not ¬φ_1(L̃), α_1(L̃,X̃,F̃,L̃′,X̃′,F̃).
            ⋮
permute(L̃,X̃,F̃,L̃′,X̃′,F̃′)  :-  not ¬φ_k(L̃), α_k(L̃,X̃,F̃,L̃′,X̃′,F̃′).
permute(L̃,X̃,F̃,L̃,X̃,F̃).

check_and_store(L̃,X̃,F̃)  :-  permute(L̃,X̃,F̃,L̃′,X̃′,F̃′),
      not subsumed(L̃′,X̃′,F̃′), store(L̃′,X̃′,F̃′).
```

Program 8.12: *Permute* and New *Check_and_Store*

8.5 Implementing Reduction

The symmetry and serializability assertions in Section 4.5 all take the form

$$p(\tilde{L},\tilde{X},\tilde{F}),\varphi(\tilde{L}) \models p(\tilde{L}',\tilde{X}',\tilde{F}'),\alpha(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}).$$

In general we are given a number of assertions, say

$$\begin{array}{c} p(\tilde{L},\tilde{X},\tilde{F}),\varphi_1(\tilde{L}) \models p(\tilde{L}',\tilde{X}',\tilde{F}'),\alpha_1(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'). \\ \vdots \\ p(\tilde{L},\tilde{X},\tilde{F}),\varphi_k(\tilde{L}) \models p(\tilde{L}',\tilde{X}',\tilde{F}'),\alpha_k(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'). \end{array}$$

Suppose that the current assertion to be proved is $p(\tilde{L},\tilde{X},\tilde{F}),\phi(\tilde{L},\tilde{X},\tilde{F}) \models \Box$, where $\phi(\tilde{L},\tilde{X},\tilde{F})$ is a sequence of non-atom constraints. According to the algorithm of Figure 6.3, In order to use a symmetry or serializability assertion, we perform the following steps:

1. Test whether $\langle \exists \tilde{X} : \phi(\tilde{L},\tilde{X},\tilde{F})\rangle$ implies $\varphi(\tilde{L})$. This establishes $G \models G'$ at $\langle 1\rangle$ in Figure 6.3.

2. Check that the assertion

$$p(\tilde{L}',\tilde{X}',\tilde{F}'),\alpha(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'),\phi(\tilde{L}) \models \Box$$

   can be directly proved ($\langle 2\rangle$ in Figure 6.3) or its lhs goal subsumed by another assertion in the table ($\langle 3\rangle$ in Figure 6.3).

By simply following the algorithm of Figure 6.3, we can implement a naive proof engine with symmetry reduction by modifying *check_and_store* as shown in Program 8.12.

Recall, however, that we only encounter assertions (goals) with ground program points. We can therefore replace each test *not* $\varphi_i(\tilde{L})$ with $\varphi_i(\tilde{L})$ for $1 \leq i \leq k$. This is because when $\tilde{L}$ is

$$\begin{array}{lcl}
\mathit{permute}(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}') & :\text{-} & \varphi_1(\tilde{L}),\alpha_1(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'). \\
\qquad\vdots & & \\
\mathit{permute}(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}') & :\text{-} & \varphi_k(\tilde{L}),\alpha_k(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'). \\
\mathit{permute}(\tilde{L},\tilde{X},\tilde{F},\tilde{L},\tilde{X},\tilde{F}). & &
\end{array}$$

Program 8.13: Second Version of *Permute*

ground, the $\langle \exists \tilde{X},\tilde{F} : \phi(\tilde{L},\tilde{X},\tilde{F})\rangle \Rightarrow \varphi_i(\tilde{L})$ if and only if $\langle \exists \tilde{X},\tilde{F} : \phi(\tilde{L},\tilde{X},\tilde{F})\rangle \wedge \varphi_i(\tilde{L})$. The updated *permute* is shown as Program 8.13.

We note here that the permutation of variable-value pairs we discussed in Section 4.5.2 is encoded in $\alpha_1,\ldots,\alpha_k$. Given a room term, say $\mathit{room}([0,1,0])$, representing a number "1," we may permute the elements of its list such that it represents different element of finite domain. For example, exchanging the 1 with the second 0 in $\mathit{room}([0,1,0])$ results in a room term $\mathit{room}([0,0,1])$ representing "2." In this way we can encode the symmetry relation in the two-process Fischer's mutual exclusion algorithm (Example 4.13 on Page 100) as

$$\begin{array}{l}
\mathit{permute}(L_1,L_2,T_1,T_2,K,L_2,L_1,T_2,T_1,K') \;:\text{-} \\
\qquad K = \mathit{room}([X,A,B]), \mathit{not}\; X = 1, K' = \mathit{room}([0,B,A]). \\
\mathit{permute}(L_1,L_2,T_1,T_2,K,L_2,L_1,T_2,T_1,K') \;:\text{-} \\
\qquad K = \mathit{room}([X,A,B]), \mathit{not}\; X = 0, K' = \mathit{room}([1,A,B]). \\
\mathit{permute}(L_1,L_2,T_1,T_2,K,L_1,L_2,T_1,T_2,K).
\end{array}$$

Here, $\mathit{room}([1,0,0])$, $\mathit{room}([0,1,0])$, and $\mathit{room}([0,0,1])$ represent "0," "1," and "2," respectively.

Unfortunately, the implementation given as Program 8.13 is in general inefficient. To see why, suppose that we have a CLP model of a concurrent program with N symmetric processes. Then this means that there are $N!$ ways to exchange the processes to obtain a symmetric program state. Hence in Program 8.13, *permute* has $k = N!$ clauses, and for each assertion in the proof tree these clauses induce $O(N!)$ table checks! In general, computing whether an encountered goal is equivalent (via symmetry) to another goal (stored in the table) is at least as hard as graph isomorphism problem [29, 107, 58, 59]. This is called *orbit problem* in [28], where it is equated to the more general problem of finding a set stabilizer of a set in a coset (SSC problem). *Orbit* is the term used for an equivalence class of states induced by symmetry.

To partially solve this problem, we use *normalization* of goals. Recall the discussion in Section 4.5 that symmetry divides the collecting semantics, and therefore goals, into a number of

$$\begin{array}{l} \mathit{check\_and\_store}(\tilde{L},\tilde{X},\tilde{F}) \ \ \text{:-}\ \ \mathit{normalize}(\tilde{L},\tilde{X},\tilde{F},\tilde{L}',\tilde{X}',\tilde{F}'), \\ \quad\quad \mathit{not\ subsumed}(\tilde{L}',\tilde{X}',\tilde{F}'),\mathit{store}(\tilde{L}',\tilde{X}',\tilde{F}'). \end{array}$$

Program 8.14: New Version of *Check_and_Store*

disjoint equivalence classes. The problem is, given a goal G, there are possibly $N!$ representatives in its equivalence class, making simple search for the correct representative highly inefficient. In order to make the number of choices smaller, we need to be more particular about possible representatives. Recall that in the proof tree we only encounter assertions $p(\tilde{l},\tilde{X},\tilde{F}),\phi(\tilde{X},\tilde{F}) \models \Box$ where $\tilde{l}$ is list of ground values. Here we restrict possible representatives to only those where $\tilde{l}$ is sorted. We therefore need only to compute the *sorted* form of goals. We may say that here we normalize a goal into its sorted form. The problem is partially mitigated since even the worst sorting procedure is still of polynomial complexity. Although we still have not escaped from the orbit problem since there are $N!$ solution in sorting a goal where $\tilde{l}$ is the list of length N of the same number, in many other cases there are only a small number of possible sorted forms. To mitigate the orbit problem even further, we limit the possible sort outcomes to a fixed number (7 in our experiments). This decreases the chance for successful subsumption due to symmetry, but it makes the engine runs faster.

The above normalization using sort function is for permutational symmetry (and cases of not-quite symmetry with some modifications). Sorting is not suitable for rotational (cyclic) symmetry (e.g. in dining philosophers' problem) since it can result in a goal not equivalent by symmetry. Fortunately, normalization is not hard for cyclic symmetry since for such problems normalization is a cyclic shift which is linear to the length of program point list $\tilde{l}$ (cf. [28]). A possibly more efficient normalization function (generating more specific range of possible representatives) that shifts until the smallest element comes first in the list can be devised.

Neither sorting nor cyclic shift explained above is necessary when using serializability assertions. For not-quite-symmetry assertions, we took a conservative approach that does not perform sorting for any goal which permutation cannot be done or is possibly irreversible.

Program 8.14 is the new version of *check_and_store*, where it calls a black box predicate *normalize* (instead of *permute*) which performs normalization according to the kind of symmetry or serializability assertion used.

8.6 Axioms

In order to obtain a proof faster, we may want to allow the user to specify a set of *axioms* to help the prover. Axioms are independently proved traditional safety assertions, which are introduced into the proof process via an application of our versatile CUT rule.

Let us now elaborate the use of axioms further by using as an example the two-process bakery mutual exclusion algorithm (Section 3.3). Representation of this problem in *init* and *trans* predicates is shown as Program 8.15, where we rewrite the disjunctions $Y = 0 \vee X < Y$ and $X = 0 \vee Y < X$ into separate rules, since this is the way CLP($\mathcal{R}$) handles disjunctions.

Previously we have proved that mutual exclusion holds in Section 5.6, where the proof uses the assumption that the variables X and Y are of positive integer domain. Unfortunately, our CLP($\mathcal{R}$) implementation uses variables in real domain. This results in nontermination of the proof process when we naively use our prover engine. To be more specific, in proving $p(2,2,X,Y) \models \Box$ (the mutual exclusion property), the following unfold sequence would be possible in the real numbers domain:

$$\begin{array}{c} p(2,2,X,Y) \models \Box \\ \downarrow \\ p(1,2,X,Y), X < Y \models \Box \\ \downarrow \\ p(1,1,X,Y), X < 0, Y = 0 \models \Box \\ \downarrow \\ p(1,0,X,Y'), X = -1, Y = 0 \models \Box \\ \downarrow \\ \vdots \end{array}$$

In fact, the above sequence is a prefix of an infinite one. However, we know that the values of x and y of the bakery algorithm are never negative. We may represent this fact using the assertion $p(L_1,L_2,X,Y) \models X \geq 0, Y \geq 0$. Now, when this assertion holds, by the CUT rule we may replace the assertion $p(1,1,X,Y), X < 0, Y = 0 \models \Box$ above with $X \geq 0, Y \geq 0, X < 0, Y = 0 \models \Box$ which holds immediately (proved by DP), and we can then stop the unfold sequence.

Timed automata models, in particular, require that the clock values are not negative. This has no consequence in a bottom-up reasoning, since clocks start with non-negative value, and

```
init(0,0,X,Y) :- X = 0,Y = 0.

trans(0,L2,X,Y,1,L2,Y+1,Y).
trans(1,L2,X,Y,2,L2,X,Y) :- Y = 0.
trans(1,L2,X,Y,2,L2,X,Y) :- X < Y.
trans(2,L2,X,Y,0,L2,0,Y).

trans(L1,0,X,Y,L1,1,X,X+1).
trans(L1,1,X,Y,L1,2,X,Y) :- X = 0.
trans(L1,1,X,Y,L1,2,X,Y) :- Y < X.
trans(L1,2,X,Y,L1,0,X,0).
```

Program 8.15: 2-Process Bakery Algorithm Problem in CLP($\mathcal{R}$)

```
negax(L̃,X̃,F̃) :- φ1(L̃,X̃,F̃).
   ⋮
negax(L̃,X̃,F̃) :- φk(L̃,X̃,F̃).

p(L̃,X̃,F̃) :- not negax(L̃,X̃,F̃),check_and_store(L̃,X̃,F̃),q(L̃,X̃,F̃).

q(L̃,X̃,F̃) :- trans(L̃,X̃,F̃,L̃',X̃',F̃'),p(L̃',X̃',F̃').
q(L̃,X̃,F̃) :- init(L̃,X̃,F̃).
```

Program 8.16: Seventh Engine

can only be incremented or reset to zero. However, we must intentionally ensures the property using an axiom in a top-down reasoning using backward CLP modeling, since clock variables are actually decremented from a goal to the next, in which it is possible to obtain a set of constraints that implies negative values for some of the clocks.

We note that the application of non-negative clock axioms is analogous to the *Border-Line* operation on time regions in an early timed automata verification framework of Yi et al. which is based on backward pre-image computation [200].

We assume that axioms are always of the form $p(\tilde{L},\tilde{X},\tilde{F}) \models \varphi(\tilde{L},\tilde{X},\tilde{F})$, where $\varphi(\tilde{L},\tilde{X},\tilde{F})$ is a sequence (conjunction) of constraints. Now assume that the negation of $\varphi(\tilde{L},\tilde{X},\tilde{F})$ is the disjunction $\phi_1(\tilde{L},\tilde{X},\tilde{F}) \vee \ldots \vee \phi_k(\tilde{L},\tilde{X},\tilde{F})$. We encode this using the *negax* predicate in Program 8.16, and checking of axiom is now encoded in the clause of p as a test of failure of the call to *negax*.

8.7 Proving Relative Safety Assertions

So far we have only presented prover engines for proving assertions of the form $p(\tilde{X}), \phi(\tilde{X}) \models \psi(\tilde{X})$, where $\phi(\tilde{X})$ and $\psi(\tilde{X})$ are constraints. We now describe the implementation techniques for proving relative safety assertions, which are of the form:

$$p(\tilde{L}_L, \tilde{X}_L, \tilde{F}_L), \varphi(\tilde{L}_L) \models p(\tilde{L}_R, \tilde{X}_R, \tilde{F}_R), \alpha(\tilde{L}_L, \tilde{X}_L, \tilde{F}_L, \tilde{L}_R, \tilde{X}_R, \tilde{F}_R).$$

A prover for this problem would essentially generate a lhs tree and at each newly encountered node, try to perform rhs unfold. Here instead of simply computing a goal from another as previously (where any assertion is of the form $G \models \square$), we propagate the whole assertion. This is because the effect of LU application on the variables $\tilde{L}_L$, $\tilde{X}_L$, and $\tilde{F}_L$ need to be reflected on the variables $\tilde{L}_R$, $\tilde{X}_R$, and $\tilde{X}_R$, respectively. Similarly, the table no longer stores just a goal (assuming all assertions are of the form $G \models \square$), but instead both lhs and rhs goals of assertions.

We show the basic skeleton of our relative safety prover as Program 8.17. The main differences with our previous provers are:

1. Instead of propagating constraints on $\tilde{L}, \tilde{X}, \tilde{F}$, we propagate constraints on $\tilde{L}_L$, $\tilde{X}_L$, $\tilde{F}_L$, $\tilde{L}_R$, $\tilde{X}_R$, and $\tilde{F}_R$.

2. Instead of global tabling, we use a list of assumed assertions, as advocated by our first algorithm of Figure 7.1. This is because we suspect that redundancy is much less likely to occur compared to hypothesis application (AP). Therefore it is more efficient to simply check for ancestor assertions than all of the previously visited assertions in the proof tree.

3. We replace direct subsumption check in the body of *check_and_store* into a proof via right unfold (RU). Here we perform the right unfolding in a depth-first manner by iterative deepening. We start with no right unfold (that is, unfold up to level 0). If subsumption does not hold, we attempt to prove by rhs unfolding up to level 1, and so on, until certain finite depth, specified by the user.

Note that in proving relative safety assertions, program points (the lists $\tilde{L}_L$ and $\tilde{L}_R$) may not be ground. Therefore the definition of *permute* for Program 8.17 should be the one given in Program 8.12, and not Program 8.13. The call to *permute* is to make use of other relative safety assertions in the proof.

```
$check\_and\_store(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A},\tilde{A}')$ :-
    $permute(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}'_L,\tilde{X}'_L,\tilde{F}'_L)$,
    $not\ right\_unfold(\tilde{L}'_L,\tilde{X}'_L,\tilde{F}'_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A})$,
    $store(\tilde{L}'_L,\tilde{X}'_L,\tilde{F}'_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A},\tilde{A}')$.

$p(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A})$ :- $check\_and\_store(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A},\tilde{A}')$,
    $q(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A}')$.

$q(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A})$ :- $init(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L)$.
$q(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A})$ :- $trans(\tilde{L}_L,\tilde{X}_L,\tilde{F}_L,\tilde{L}'_L,\tilde{X}'_L,\tilde{F}'_L), p(\tilde{L}'_L,\tilde{X}'_L,\tilde{F}'_L,\tilde{L}_R,\tilde{X}_R,\tilde{F}_R,\tilde{A})$.
```

Program 8.17: Relative Safety Prover

8.8 Implementing Dynamic Summarization

We now explain how we implement the dynamic summarization introduced in Chapter 6. We note again that dynamic summarization is applied only on non-cyclic fragments of programs. Our implementation, however, can analyze sequential programs with loops. Here we view terminating loops with counter as acyclic because each iteration of a terminating loop is different from one another by the loop counter. We implement a version of dynamic summarization which uses constraint deletion as described in Section 6.2.4. Because the actual implementation is rather involved, here we will not display the concrete code but instead provide an overview of the techniques used.

In order to perform constraint deletion, our CLP implementation maintains a list of explicit constraints collected on an execution trace. Other than this, it also maintains a *flag list* of the same length as the constraints list. An element of the flag list at a particular position denotes whether the constraint at the same position can be deleted or not. When the element is "_," it means that the constraint can be deleted. An "o" denotes that the constraint cannot be deleted. The whole contents of the flag list is initially set to "_." Whenever a new constraint is added in a left unfold step, the flag list is lengthened by adding a "_."

We implement a procedure *filter* to implement the functions $cdel_1$, $cdel_2$, $cdel_{3(b)}$, and $cdel_{3(c)}$, of Section 6.2.4. *Filter* linearly scans the constraint list to find constraints that are to be kept to ensure that the whole conjunction still preserves the unsatisfiability or the desired postcondition (for $cdel_1$ and $cdel_2$), subsumption (for $cdel_{3(b)}$), or provability by stronger assertion (for $cdel_{3(c)}$). It does this by evaluating the whole constraint list without the constraint at position i. If the desired condition still holds, it removes this constraint from the list and advances to position $i+1$.

Otherwise, it retains the constraint and mark the flag list at the same position with an “o.”

Recall that the function $cdel_{3(\mathrm{a})}$ in Section 6.2.4 basically returns a potential summarization of an assertion in the proof tree wrt. the summarizations of the assertions to which it has been left-unfolded. It is indirectly implemented also by *filter*. We note that the flag list of a child node is actually an extension of the flag list of its parent. Modifying the flag list at the child level (as an implementation of $cdel_1$, $cdel_2$, $cdel_{3(\mathrm{b})}$, and $cdel_{3(\mathrm{c})}$) also updates the flag list of the parent. This is how we propagate the deletable constraints of a node to its ancestors.

Computing the final summarization of an assertion is done by unification of the flag lists returned from the processing of its immediate left-unfold children. Since the unification of “o” and “_” is “o,” any constraint that is required in a particular unfold branch will also be marked as required for the assertion.

Finally, we note that for efficiency we always retain constraints of the form $X = X'$, say of a statement that does not modify a variable x, such that they need not be considered explicitly. This is also because they do not contribute to unsatisfiability of a goal or preservation of postcondition.

8.9 On the Implementation of Arrays

We do not implement array reasoning in our current prover prototype. Handling array requires more complete and complex constraint solving. Here we propose ways to propagate array constraints in future versions of the prototype with respect to the following two problems:

1. Array reference $A[I]$ is not parsed by CLP($\mathcal{R}$). Therefore, to represent the value of array reference $A[I]$ using a variable, say X, and carry the relation in the proof tree. We denote $A[I] = X$ using the term $aref(X,A,I)$, called an *array reference term*. In traversing a path in the proof tree, we accumulate array reference terms in a list.

   Program 8.18 contains *init* and *trans* predicates of the bubble sort program (Program 3.5) given in Section 3.1.5. It is a rather straightforward translation of Program 3.6 (Page 52), where the clause κ_i in Program 3.6 corresponds to clause λ_i in Program 8.18.

   Notice that we use array reference terms in clauses λ_8, λ_9, and λ_{10}, since array references of the form $A[J]$ and $A[J+1]$ appear in clauses κ_8, κ_9 and κ_{10} of Program 3.6.

2. Array update expression $\langle A,I,X\rangle$ cannot be parsed by CLP($\mathcal{R}$) as well. To represent an array update expression $\langle A,I,X\rangle$ we use the *array update term* $aupd(A,I,X)$. The use is demonstrated in clause λ_{10} of Program 8.18.

```
init(8,A,I,J,N,A,N,L).                                                              λ1
trans(0,A,I,J,N,Af,Nf,L,1,A,0,J,N,Af,Nf,L).                                         λ2
trans(1,A,I,J,N,Af,Nf,L,8,A,I,J,N,Af,Nf,L) :- I ≥ N−1.                              λ3
trans(1,A,I,J,N,Af,Nf,L,2,A,I,J,N,Af,Nf,L) :- I < N−1.                              λ4
trans(2,A,I,J,N,Af,Nf,L,3,A,I,0,N,Af,Nf,L).                                         λ5
trans(3,A,I,J,N,Af,Nf,L,7,A,I,J,N,Af,Nf,L) :- J ≥ N−1−I.                            λ6
trans(3,A,I,J,N,Af,Nf,L,4,A,I,J,N,Af,Nf,L) :- J < N−1−I.                            λ7
trans(4,A,I,J,N,Af,Nf,L,6,A,I,J,N,Af,Nf,[aref(X,A,J+1),aref(Y,A,J)|L]) :-
    X > Y.                                                                          λ8
trans(4,A,I,J,N,Af,Nf,L,5,A,I,J,N,Af,Nf,[aref(X,A,J+1),aref(Y,A,J)|L]) :-
    X ≤ Y.                                                                          λ9
trans(5,A,I,J,N,Af,Nf,L,6,A',I,J,N,Af,Nf,[aref(X,A,J+1),aref(Y,A,J)|L]) :-
    A' = aupd(aupd(A,J+1,Y),J,X).                                                   λ10
```

Program 8.18: *Init* and *Trans* of Bubble Sort CLP Model

Both array update and reference terms can be accumulated first and then solved at the last obligations in the proof tree when no more left unfold (LU) is possible, or they can be solved at various points in the proof tree. We may solve them using implementations of (AIP) and (SEP) principles introduced in Section 5.9.

8.10 Experimental Results

We implement our proof engines as regular CLP($\mathcal{R}$) [110] programs, making use of its meta-level facilities. In this section we discuss four kinds of experiments using our prototypes: proving traditional safety using intermittent predicate abstraction, proving of relative safety assertions, proving of traditional safety with reduction using symmetry and serializability assertions, and proving of traditional safety assertions using dynamic summarization. The experiments reported in this section are all conducted on on a 2.8 GHz Pentium 4 machine with 512MB of RAM running Linux, except for TSA problems in Section 8.10.3, which we ran on Pentium 4 Xeon cluster with 2.0GB RAM and minimum CPU clock speed set to 2.0 GHz.

8.10.1 Experiments on Intermittent Abstraction

We first show an example that demonstrates, in a predicate abstraction setting, that intermittent abstraction requires fewer predicates than when abstraction is applied at every point in the proof tree. Let us consider a looping program written in C (Program 8.19). We note that the C program can be straightforwardly translated into its CLP model, similar to the way we translate programs

```
int main() {
   int i=0, j, x=0;
   while (i<50) {
      i++; j=0;
      while (j<10) { x++; j++; }
      while (x>i) { x--; }
   } }
```

Program 8.19: Program with Loop

in our simple programming language into CLP in Chapter 3. The program's postcondition $x \geq 50$ can be proved by providing an invariant $x = i \wedge i < 50$ before the first statement of the loop body of the outer while loop. For predicate abstraction, we supplied the predicates $x = i$, $i < 50$, and respectively their negations $x \neq i$, $i \geq 50$ for that program point to our verifier. We then ran our prover engine with the intermittent abstraction. As the result, the execution finished in less than 0.01 seconds. When we did not provide an abstraction, the execution finished in 20.34 seconds. Here intermittent abstraction required fewer predicates: We also ran the same program with BLAST and provided the predicates $x = i$ and $i < 50$ (BLAST automatically also included their negations). BLAST finished in 1.33 seconds, and in addition, it also produced 23 other predicates through refinements. Running it again with all these predicates given, BLAST finished in 0.28 seconds.

Further, we also ran our prover on a "sequential" version of the bakery mutual exclusion algorithm (Program 3.14 in Section 3.3.1), whose two-process version is shown as Program 8.20. In Program 8.20 we use if the conditions `__BLAST_NONDET` which is compiled by BLAST into nondeterministic branching. Program 8.20, including the nondeterministic branching, is also straightforwardly translated into CLP. For this experiment we performed runs with two, three, and four process versions of the sequentialized bakery algorithm. When N is the number of processes, each of the version has the N variables pc_i, where $1 \leq i \leq N$, each denoting the program point of process i as in Program 3.14. Pc_i can only take a value from $\{0,1,2\}$. Each of the two, three, or four process versions also has N variables x_i, each denoting the "ticket number" (x or y in Program 3.14) of a process.

Here we needed an abstraction to terminate the analysis since the bakery algorithm is has an infinite state space. Here we verified mutual exclusion, that is, no two processes are in the critical section ($pc_i = pc_j = 2$ when $i \neq j$) at the same time. Here we performed three sets of runs, each consisting of runs with two, three and four processes. In all three sets, we use a basic set of predicates: $x_i = 0$, $x_i \geq 0$, $pc_i = 0$, $pc_i = 1$, $pc_i = 2$, where $i = 1, \ldots, N$ and N the number

```
    int main()
    {
⟨0⟩ int pc1=0, pc2=0;
    unsigned int x1=0, x2=0;
⟨1⟩ while (1) {
⟨2⟩    if (pc1==1 || pc2==1) {
⟨3⟩       /* Abstraction point 1 */; }
⟨4⟩    if (pc1==0 || pc2==0) {
⟨5⟩       /* Abstraction point 2 */;
       } else if (pc1==2 && pc2==2) {⟨6⟩ ERROR: }
⟨7⟩    if (__BLAST_NONDET) {
⟨8⟩       if (pc1==0) {
⟨9⟩          x1=x2+1; pc1=1;
          } else if (pc1==1 &&
             (x2==0 || x1<x2)) {
⟨10⟩         pc1=2;
          } else if (pc1==2) {
⟨11⟩         pc1=0; x1=0;
          }
       } else {
⟨12⟩      if (pc2==0) {
⟨13⟩         x2=x1+1; pc2=1;
          } else if (pc2==1 &&
             (x1==0 || x2<x1)) {
⟨14⟩         pc2=2;
          } else if (pc2==2) {
⟨15⟩         pc2=0; x2=0;
    }}}}
```

Program 8.20: Sequential 2-Process Bakery

of processes, and also their negations.

- **Set 1: Use of abstraction at every state with full predicate set**. We perform abstraction at every state encountered during search. In addition to the basic predicates, we also require the inclusion of the predicates shown in Table 8.1 (a) (and their negations) to avoid *spurious counterexamples*, which are counterexample traces resulting from coarse abstraction which do not exist in the actual run of the program.

- **Set 2: Intermittent abstraction with full predicate set**. We use intermittent abstraction on our prototype implementation. We abstract only when for some process i, $pc_i = 1$ holds. The set of predicates is as in the first set.

- **Set 3: Intermittent abstraction with reduced predicate set**. We use intermittent abstraction on our tabled CLP system. We only abstract whenever there are $N-1$ processes at program point 0 (in the two-process sequential version this means either $pc_1 = 0$ or

Bakery-2	$x_1 < x_2$
Bakery-3	$x_1 < x_2, x_1 < x_3, x_2 < x_3$
Bakery-4	$x_1 < x_2, x_1 < x_3, x_1 < x_4$ $x_2 < x_3, x_2 < x_4, x_3 < x_4$

(a) Additional Predicates

	Time (in Seconds)			
	CLP with Tabling			BLAST
	Set 1	Set 2	Set 3	
Bakery-2	0.02	0.01	<0.01	0.17
Bakery-3	0.83	0.14	0.09	2.38
Bakery-4	131.11	8.85	5.02	78.47

(b) Timing Comparison

Table 8.1: Results of Experiments Using Abstraction

$pc_2 = 0$). For a N-process bakery algorithm, we only need the basic predicates and their negations without the additional predicates shown in Table 8.1 (a).

We also compare our results with BLAST. We supplied the same set of predicates that we used in the first and second sets to BLAST. Again, in BLAST we do not have to specify their negations explicitly. Interestingly, for the four-process bakery algorithm BLAST requires even more predicates to avoid refinement, which are $x_1 = x_3 + 1$, $x_2 = x_3 + 1$, $x_1 = x_2 + 1$, $1 \leq x_4$, $x_1 \leq x_3$, $x_2 \leq x_3$ and $x_1 \leq x_2$. We suspect this is due to the fact that precision in predicate abstraction-based state-space traversal depends on the power of the underlying theorem prover. We have BLAST generate these additional predicates it needs in a pre-run, and then run BLAST using them. Here since we do not run BLAST with refinement, the *lazy abstraction* technique [97] has no effect, and BLAST uses all the supplied predicates to represent any abstract state.

For these problems, using our intermittent abstraction with CLP tabling is also markedly faster than both full predicate abstraction with CLP and BLAST. We show our timing results in Table 8.1 (b) (smallest recorded time of three runs each).

The first set and BLAST both run with abstraction at every visited state. The timing difference between them and second and third sets shows that performing abstraction at every visited state is expensive. The third set shows further gain over the second when we understand some intricacies of the system and able to employ abstraction more carefully.

8.10.2 Experiments on Relative Safety

Here we discuss the run results of our prototype implementations for proving relative safety assertions, which we have discussed in Section 8.7.

Experimental results in proving relative safety assertions are shown in Table 8.2. In *Problem-Name-N*, N denotes the number of processes, except for *Prod/Cons-N* where N denotes that there are N produce and consume operations. For each problem we verify a number of relative

Problem	No. of Assertions	Right Iter. Depth Bound	Nodes	Hypothesis Applications	Time (s)
Bakery-2	1	∞	9	9	0.00
Bakery-3	2	∞	44	44	0.04
Bakery-4	3	∞	147	147	0.30
Bakery-5	4	∞	424	424	2.11
Bakery-6	5	∞	1145	1145	13.23
Bakery-7	6	∞	3486	3486	81.38
Philosopher-3	1	∞	19	19	0.01
Philosopher-4	1	∞	25	25	0.01
Priority	1	3	39	20	0.05
Priority	1	4	30	17	0.05
Priority	1	5	23	13	0.05
Szymanski-2	8	3	1470	63	7.53
Szymanski-2	8	4	1276	71	13.21
Szymanski-2	8	5	1107	87	20.35
Prod/Cons-2	2	10	17	12	0.59
Prod/Cons-3	2	10	42	24	2.43
Prod/Cons-4	2	10	303	134	11.37
Prod/Cons-5	2	10	1487	619	70.84

Table 8.2: Relative Safety Proof Experimental Results

safety assertions. The “Nodes” and “Time” numbers are total space and time in proving all of the assertions of each problem. “Right Iter. Depth Bound” column in the table represents the maximal right unfold depth.

As we have discussed in Section 8.5, the total number of symmetry assertions to be proved in fully symmetric systems is of order factorial to the number of processes in the worst case. However, it is actually enough to prove just a subset of these assertions, which are those that constitute exchanges of two adjacent positions, since other assertions can be immediately derived from this subset. The size of this subset is linear to the number of processes, and since the number of transitions is also linear to the number of processes, we expect the proof size of symmetry for fully permutable systems to be of cubic order to the number of processes. To see this, first note that for fully symmetric programs, we only need one level of left and right unfolds. In effect the proof of each assertion is a comparison of transitions with one another, which is of quadratic complexity to the number of processes. This is then multiplied by the number of assertions obtaining the cubic order. As we see from Table 8.2, however, the runs of bakery algorithm has more than cubic complexity. This is because the number of transitions itself increases more than linearly to the number of processes due to need of encoding disjuncts in **await** guards as separate CLP clauses. For dining philosophers, there is always only one assertion, which is one

Problem	Our Implementation				Delzanno-Podelski	XMC/RT	
	No Assertion		W/ Assertion				
	# Stored	Time	# Stored	Time	# Facts	# Answers	Time
Bakery-2	15	0.00	8	0.00	13		
Bakery-3	296	0.08	45	0.01	109		
Bakery-4	4624	5.90	191	0.21	963		
Bakery-5	∞	∞	677	2.76			
Bakery-6	∞	∞	2569	45.82			
Bakery-7	∞	∞	11865	971.84			
Peterson-2	105	0.03	10	0.00			
Peterson-3	20285	107.53	175	0.14			
Peterson-4	∞	∞	3510	11.38			
Lamport-2	143	0.01	72	0.01			
Lamport-3	4255	0.89	707	0.25			
Lamport-4	∞	∞	5626	4.44			
Priority	8	0.00	8	0.00			
Szymanski-2	240	0.08	84	0.02			
Szymanski-3	10883	35.43	3176	2.91			
Philosopher-3	882	0.46	553	0.30			
Philosopher-4	4293	24.44	2783	8.48			
Prod/Cons-2	64	0.00	43	0.00			
Prod/Cons-3	104	0.01	59	0.01			
Prod/Cons-4	154	0.01	75	0.01			
Prod/Cons-5	214	0.02	91	0.01			
Prod/Cons-10	664	0.10	171	0.02			
Prod/Cons-20	2314	1.90	331	0.04			
Fischer TSA-4	875	1.66	72	0.03		632	0.82
Fischer TSA-5	6872	203.47	165	0.13		6330	8.91
Fischer TSA-6	∞	∞	325	0.42		75972	187.16
Fischer TSA-7	∞	∞	591	1.41		∞	∞
Fischer TSA-8	∞	∞	1016	6.17		∞	∞
Fischer TSA-9	∞	∞	1649	37.79		∞	∞
Fischer TSA-10	∞	∞	2536	322.76		∞	∞
Fischer TSA-11	∞	∞	3759	3176.60		∞	∞
Philosopher TSA-3	147	0.16	89	0.10		422	0.16
Philosopher TSA-4	640	2.40	322	1.11		∞	∞
Philosopher TSA-5	5776	188.84	2340	58.29		∞	∞

Table 8.3: Traditional Safety Proof Experimental Results

position right circular shift since other cyclical transpositions can be derived from this. Hence for rotational symmetry, the cost is of order quadratic to the number of processes (just the number of transitions comparison), which is confirmed by the table. For simple priority mutual exclusion and 2-process Szymanski's algorithm, we also notice a decrease in the number of left tree size when we increase the right unfold depth bound, although the execution time also increases most likely due to the need to examine larger program.

8.10.3 Experiments on Traditional Safety with Reduction

Our last experiment is on automatically proving traditional safety assertions of the form $G \models \Box$ with or without reduction via relative safety assertions.

Problem	% Reduction	
Type	Space	Time
Bakery	76%	78%
Peterson	95%	99.9%
Lamport	67%	65%
Szymanski	68%	83%
Philosopher	36%	53%
Prod/Cons-10 and *-20*	87%	94%
Fischer TSA	95%	99%
Philosopher TSA	50%	53%

Table 8.4: Percent Reduction

The results are shown in Table 8.3. In the table, "# Stored" denotes the number of assertions stored in the table, as an indicator of the size of the search space, and times are in seconds. We ran the bakery, Peterson's, Lamport's fast mutual exclusion, Szymanski's, and the TSA version of the Fischer's algorithms proving mutual exclusion. Note that we do not prove the symmetry assertions of some of the problems (e.g., Szymanski-3). For the dining philosophers' problems (both the program and TSA versions), we prove that there cannot be more than $N/2$ philosophers simultaneously eating. For the producer-consumer problem (Program 4.26), each $pro_i()$ increments a variable x, and $con_j()$ decrements it. Here we verify that the value of x can never be more than $2N$.

Bakery algorithm has infinite reachable states, and therefore cannot be handled by finite-state model checkers. Here we compare our search space with the results of the CLP-based system of Delzanno and Podelski as reported in [46]. As also noted by Delzanno and Podelski, the problem does not scale well to larger number of processes, but using symmetry, we have pushed its verification limit to 7 processes without abstraction.

We compare our results for TSA problems with the run of similar example on XMC/RT [156]. The XMC/RT system is implemented on XSB [175, 176], and utilizes built-in tabling based on SLG resolution, as well as a DBM library. In our experiments, we only considered reachability analysis in XMC/RT. XMC/RT has a prover engine that generates all possible execution traces (answers) in a bottom-up manner and checks for violating traces. With XMC/RT we count the number of such traces generated. For the Fischer's algorithm, without symmetry reduction the space complexity of our implementation seems to be similar to XMC/RT, although the time complexity seems to be much worse. Of course, our runs can be expedited by symmetry reduction, as shown in the table.

In Table 8.4 we summarize the effectiveness of the use of a variety of relative safety assertions. The use of symmetry assertion effectively reduces the search space of perfectly symmetric problems (bakery, Peterson's, Lamport's fast mutex, TSA Fischer's algorithm, dining philosophers, both program and TSA versions). Notice also that the reduction for Szymanski's algorithm is competitive with perfectly symmetric problems, showing that "not-quite" symmetry reduction is worth pursuing. The use of rotational symmetry in both versions of the dining philosophers' problem is, expectedly, less effective due to the fact that circular shift is more restrictive than full permutation. We also note that we managed to obtain a substantial reduction of search space for the producer/consumer problem. Reduction in time roughly corresponds to those of search space.

We note here that the TSA verification tool RED also has symmetry reduction capability, both for full permutational symmetry and cyclic symmetry. It has exceeded our result for Fischer's algorithm run at 13 processes [197]. However, RED loses precision for problems with cyclic structure [198]. In contrast, our engine does not lose precision due to symmetry.

Finally, comparing Table 8.2 and 8.3, it can be seen that the proof of relative safety assertions are no easier than the proof of traditional safety assertions. This is because of the need to perform rhs unfold when proving relative safety. We may consider future optimizations for proving relative safety, such as storing of right unfold goals in a separate table for reuse without redoing the right unfolding.

8.10.4 Experiments on Dynamic Summarization

We also implement a prototype which is optimized using dynamic summarization. For the experiments, we use sample programs from *worst-case execution time* (*WCET*) benchmark suites, where we prove (via discovery) a timing bound of each program.

The experimental results are shown in Table 8.5 (time is in seconds). We note that the encoder and decoder problems are taken from ADPCM encoder and decoder appeared in [167]. The sqrt and qurt programs are from SNU RT Benchmark Suite [185], and the janne_complex program is from Mälardalen Benchmark Suite [132]. We also ran our prover on a generic bubble sort program and another version of bubble sort where each element of the array has a binary domain.

From Table 8.5, the amount of reduction obtainable by dynamic summarization seems to be inversely correlated to the amount of unsatisfiable goals encountered in the proof tree. The encoder example has a structure with interdependence of sequences of if statements. In the binary bubble sort, the limitation in the possible elements produced limitations on the number possible

Problem	No Optim.		Optimized		% Space (Time)
	Spc.	Time	Spc.	Time	Reduction
encoder	494	0.91	252	0.41	48.99% (54.95%)
decoder	344	0.31	38	0.02	88.95% (93.55%)
sqrt	923	4.25	91	0.38	90.14% (91.06%)
qurt	1104	14.47	273	2.52	75.27% (82.58%)
janne_complex	1517	17.93	410	2.13	72.97% (88.12%)
bubble sort (5)	1034	94.49	58	0.58	94.39% (99.39%)
binary bubble sort (4)	381	10.88	170	4.00	55.38% (63.24%)

Table 8.5: Experimental Results of Dynamic Summarization

swaps. This in turn increased the number of unsatisfiable goals. For both of these examples, the amount of reductions obtained were not as high as the other examples, for which dynamic summarization performed well.

8.11 Related Work

A tabling mechanism exists for logic programs, which is called *SLG*-resolution [26] and is implemented in the XSB logic programming system [175]. The XSB system tables both the formula and the answers of the formula. In our tabling mechanism, we only keep derived formulas without keeping its answer constraints.

To mitigate the orbit problem, Emerson and Trefler proposes an approach using *generic representatives* [54], for example, in a symmetric mutual exclusion algorithm with process locations N, T and C (representing *non-critical*, *trial* and *critical section*, respectively), the states (N,N,T,C), (N,C,T,N) and (T,N,N,C) are equivalent and can be represented in a generic manner as $(2N,1T,1C)$. This is exploited further by Emerson and Wahl to mitigate the orbit problem [62]. Emerson and Wahl propose a transformation of the program to be verified into a program with counters: For example, a transition of any concurrent process from N to T is transformed into a new global transition with the decrement of the counter n_N and the increment of the counter n_T of the new program, where n_N is the total number of processes in state N, while n_T is the total number of processes in state T. The property to be verified is also similarly transformed. The transformed program is shown to be bisimilar to the original program. In this way, the states of the transformed program are exactly the generic representatives, and hence the computing of representative using orbit relation during traversal is not necessary.

We also mention the work of Tang et al. [190] is on using symmetry for unbounded SAT-based model checker. The model checking approach is however unique:

1. Before the model checking process, for each orbit, a *symmetry-breaking constraint* is generated. When it is conjuncted with a newly encountered frontier, this symmetry-breaking constraint will filter out those states that are necessarily impossible to be representatives. Hence, what are stored in the BDDs are all the possible representatives. For example the symmetry-breaking constraint would specify that only states with $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$ can be representative for all permutations of x_1, x_2, x_3, where exactly one of them has the value 1, 2, or 3.

2. The model checker only stores those states that are possible to be a representative of its orbit.

The symmetry-breaking constraint can be adjusted to be more constrained, resulting in more reduction in state space (but possible harder to compute representatives), or less constrained, resulting in more states stored in the BDD.

Chapter 9

Conclusion

In this thesis we propose a CLP-based framework that accommodates both program analysis and program verification approaches. Our framework is centered on a general verification condition computation algorithm which performs abstraction intermittently. This allows for compositionality and simpler yet accurate abstraction than normal abstract interpretation. Our first primary contribution is that the algorithm is optimized between the abstraction points using a dynamic summarization technique.

There are three formal foundations of our framework including:

1. modeling of programs and high-level specifications in CLP where we model the computer memory as an array,
2. assertions to specify traditional safety (invariance) including recursive data structure properties as well as relative safety (non-behavioral/structural properties), and
3. a proof method for proving the assertions.

Our framework handles traditional safety proof including proof of recursive pointer data structures. The precision of the verification is helped by intermittent abstraction. Our other contribution is the proof of relative safety assertions which accommodates symmetry properties not handled by existing approaches, as well as commutativity and serializability properties. We use these relative safety properties for reduction in proof size.

We also provide a general algorithm for program analysis and verification based on our proof method. We discussed the implementations of some simpler variants of our general algorithm in CLP($\mathcal{R}$) and provide the experimental results.

As future work, our framework is to be extended with the verification of liveness properties such as termination of programs. This is made possible by the fact that the proof tree in our framework also represents possible execution traces of the program. A feasible approach to liveness verification is by a form of discovery of well-foundedness in the proof tree. In handling liveness, the concept of *fairness* will become important. In the modeling side, a possible technique to represent fairness in the CLP model is by providing a variable representing a counter which has an positive indefinite but not infinite value. Since the counter is always decreased yet stays positive, it will eventually reach 0, which, when detected, affects further execution of the program.

We note that the above proposed modeling of fairness can also be used to model Büchi automata in CLP, since Büchi automata, in addition to modeling behavior (state transitions), also model eventuality of states. This would provide a way to model not only behavioral specifications such as timed automata and statecharts, but also specifications with inherent liveness such as *live sequence charts* (*LSCs*) [90]. In this way we would be able to cover more formal specification languages. The simulation of specifications in these languages is important for systems development, and it therefore can also be a topic for future work.

Throughout this thesis, we have always assumed that abstraction is provided by the user. As future work, it is also possible to consider automated generation of abstractions. One possible direction is work on loop invariant discovery. The idea is to start from the execution context of a loop. The algorithm iteratively generalizes the context until it finds a suitable loop invariant.

In Chapter 6 we have discussed that it is also possible to use the dynamic summarization technique to infer information from program. This can be extended in the future toward an algorithm for general resource (e.g., time) analysis. Whereas the technique presented in this thesis can discover at most a bound on resource usage, this technique can be advanced to discover an exact bound on resource usage.

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):273–303, May 2001.

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

[5] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Graduate Texts in Computer Science. Springer, 2nd edition, 1997.

[6] A. Armando, C. Castellini, and J. Mantovani. Software model checking using linear constraints. In J. Davies, W. Schulte, and M. Barnett, editors, *6th ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 209–233. Springer, 2004.

[7] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In PLDI2001 [159], pages 203–213. SIGPLAN Notices 36(5).

[8] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 27(2):314–343, 2005.

[9] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *7th TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.

[10] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *8th SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001.

[11] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In POPL2002 [161], pages 1–3. SIGPLAN Notices 37(1).

[12] G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In Cleaveland [31], pages 163–177.

[13] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. A. Henzinger, editors, *8th CAV*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer, 1996.

[14] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2004.

[15] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[16] P. Bhaduri and S. Ramesh. Model checking of Statechart models: Survey and research directions. *CoRR*, cs.SE/0407038, 2004.

[17] T. Bienmüller, U. Brockmeyer, W. Damm, G. Döhmen, C. Eßmann, H.-J. Holberg, H. Hungar, B. Josko, R. Schlör, G. Wittich, H. Wittke, G. Clements, J. Rowlands, and E. Sefton. Formal verification of an avionics applications using abstraction and symbolic model checking. In F. Redmill and T. Anderson, editors, *7th Safety-Critical Systems Symposium*, pages 150–173. Springer, 1999.

[18] T. Bienmüller, W. Damm, and H. Wittke. The STATEMATE verification environment—making it real. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 561–567. Springer, 2000.

[19] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[20] G. S. Boolos and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 3rd edition, 1989.

[21] A. Bossi, editor. *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR '99, Venezia, Italy, September 22-24, 1999, Selected Papers*, volume 1817 of *Lecture Notes in Computer Science*. Springer, 2000.

[22] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, 1975.

[23] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[24] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *10th CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1990.

[25] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[26] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[27] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In R. Cousot, editor, *10th SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 463–482. Springer, 2003.

[28] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In Hu and Vardi [104], pages 147–158.

[29] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [32], pages 450–462.

[30] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003.

[31] R. Cleaveland, editor. *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[32] C. Courcoubetis, editor. *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993.

[33] P. Cousot. Semantic foundations of program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 10, pages 303–343. Prentice-Hall, 1981.

[34] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.

[35] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282. ACM Press, 1979.

[36] S. Craciunescu. Proving the equivalence of CLP programs. In Stuckey [188], pages 287–301.

[37] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

[38] W. Damm and J. Helbig. Linking visual formalisms: A compositional proof system for statecharts based on symbolic timing diagrams. In E.-R. Olderog, editor, *PROCOMET '94*, pages 267–286. North-Holland, 1994. IFIP Transactions A-56.

[39] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS '97*, volume 1536 of *Lecture Notes in Computer Science*, pages 186–238. Springer, 1998.

[40] D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In Zuck et al. [202], pages 310–324.

[41] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *11th CAV*, number 1633 in Lecture Notes in Computer Science, pages 160–171. Springer, 1999.

[42] A. David, M. O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In R.-D. Kutsche and H. Weber, editors, *5th FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2002.

[43] M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 2nd edition, 1994.

[44] N. Day. A model checker for Statecharts (linking CASE tools with formal methods). Master's thesis, Department of Computer Science, University of British Columbia, 1993. Technical report TR-93-95.

[45] G. Delzanno and A. Podelski. Model checking in CLP. In Cleaveland [31], pages 223–239.

[46] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[47] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[48] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1990.

[49] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as hardware design aid. In *ICCD '92*, pages 522–525. IEEE Computer Society Press, 1992.

[50] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *12th TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[51] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*, pages 175–184. IEEE Computer Society Press, 2000.

[52] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2–3), 2004.

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

[54] E. A. Emerson. From asymmetry to full symmetry: New techniques for symmetry reductions in model checking. In L. Pierre and T. Kropf, editors, *10th CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 1999.

[55] E. A. Emerson, J. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *15th LICS*, pages 121–131. IEEE Computer Society Press, 2000.

[56] E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In E. Brinksma, editor, *3rd TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 1997.

[57] E. A. Emerson and K. S. Namjoshi, editors. *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8–10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.

[58] E. A. Emerson and A. P. Sistla. Model checking and symmetry. In Courcoubetis [32], pages 463–478.

[59] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[60] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.

[61] E. A. Emerson and R. J. Trefler. Model checking real-time properties of symmetric systems. In L. Brim, J. Gruska, and J. Zlatuska, editors, *23rd MFCS*, volume 1450 of *Lecture Notes in Computer Science*, pages 427–436. Springer, 1998.

[62] E. A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In D. Geist and E. Tronci, editors, *12th CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 2003.

[63] R. Eshuis, D. N. Jansen, and R. Wieringa. Requirements-level semantics and model checking of object-oriented statecharts. *Requirements Engineering Journal*, 7:243–263, 2002.

[64] K. Etessami and S. K. Rajamani, editors. *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6–10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[65] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. R. Ramakrishnan, and U. Ultes-Nitsche, editors, *2nd VCL*, pages 85–96, 2001.

[66] C. Flanagan. Automatic software model checking using CLP. In P. Degano, editor, *12th ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2003.

[67] C. Flanagan. Verifying commit-atomicity using model-checking. In S. Graf and L. Mounier, editors, *11th SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[68] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *31st POPL*. ACM Press, 2004.

[69] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In POPL2005 [162], pages 110–121.

[70] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *16th PLDI*, pages 234–245. ACM Press, May 2002. SIGPLAN Notices 37(5).

[71] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *17th PLDI*, pages 338–349. ACM Press, 2003.

[72] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. The Jini Technology Series. Addison-Wesley, 1999.

[73] E. C. Freuder, editor. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachussets, USA, August 19–22, 1996*, volume 1118 of *Lecture Notes in Computer Science*. Springer, 1996.

[74] L. Fribourg. Automatic generation of simplification lemmas for inductive proofs. In V. A. Saraswat and K. Ueda, editors, *ISLP 1991*, pages 103–116. MIT Press, 1991.

[75] L. Fribourg. Constraint logic programming applied to model checking. In Bossi [21], pages 30–41.

[76] S. Garfinkel. History's worst software bugs. URL http://www.wired.com/software/coolapps/news/2005/11/69355, November 2005.

[77] R. Giacobazzi, editor. *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26–28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.

[78] S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using JACK. In R. Paul and C. Meadows, editors, *4th HASE*. IEEE Computer Society Press, 1999.

[79] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.

[80] S. Graf and H. Saïdi. Construction of abstract state graphs of infinite systems with PVS. In Grumberg [82], pages 72–83.

[81] S. Graf and M. I. Schwartzbach, editors. *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*. Springer, 2000.

[82] O. Grumberg, editor. *Computer Aided Verification: 9th International Conference, CAV '97, Haifa, Israel, June 1997, Proceedings*, number 1254 in Lecture Notes in Computer Science. Springer, 1997.

[83] B. Guo, N. Vaccharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *21st PLDI*, pages 256–265. ACM Press, 2007.

[84] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *18th RTSS*, pages 230–239. IEEE Computer Society Press, 1997.

[85] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, November 1999.

[86] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In POPL2005 [162], pages 310–323.

[87] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[88] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[89] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[90] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[91] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. Technical Report CS95-31, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, 76100 Rehovot, Israel, October 1995.

[92] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[93] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the Statemate Approach*. McGraw-Hill, 1998.

[94] N. C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2*, September 1992.

[95] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *5th PLDI*, pages 249–260. ACM Press, July 1992. SIGPLAN Notices 27(7).

[96] M. Hendriks, G. Behrmann, K. G. Larsen, and F. W. Vaandrager. Adding symmetry reduction to Uppaal. In K. G. Larsen and P. Niebert, editors, *1st FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.

[97] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In POPL2002 [161], pages 58–70. SIGPLAN Notices 37(1).

[98] T. A. Henzinger, P.-H. Lo, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In Grumberg [82], pages 460–463.

[99] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[100] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[101] G. J. Holzmann. *The* SPIN *Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[102] J. Hsiang and M. Srivas. Automatic inductive theorem proving using Prolog. *Theoretical Computer Science*, 54(1):3–28, 1987.

[103] J. Hsiang and M. K. Srivas. A PROLOG framework for developing and reasoning about data types. In H. Ehrig, C. Floyd, M. Nivat, and J. W. Thatcher, editors, *1st TAPSOFT Vol. 2*, volume 186 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 1985.

[104] A. J. Hu and M. Y. Vardi, editors. *Computer Aided Verification: 10th International Conference, CAV '98. Vancouver, BC, Canada, June/July 1998. Proceedings.*, number 1427 in Lecture Notes in Computer Science. Springer, 1998.

[105] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[106] O. H. Ibarra, P. C. Diniz, and M. C. Rinard. On the complexity of commutativity analysis. In J. Cai and C. K. Wong, editors, *2nd COCOON*, volume 1090 of *Lecture Notes in Computer Science*, pages 323–332. Springer, 1996.

[107] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

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

[109] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46, October 1998.

[110] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[111] J. Jaffar, A. Santosa, and R. Voicu. A CLP proof method for timed automata. In *25th RTSS*, pages 175–186. IEEE Computer Society Press, 2004.

[112] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. To be published in the Proceedings of AAAI '08.

[113] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP method for compositional and intermittent predicate abstraction. In Emerson and Namjoshi [57], pages 17–32.

[114] J. Jaffar, A. E. Santosa, and R. Voicu. Relative safety. In Emerson and Namjoshi [57], pages 282–297.

[115] D. N. Jayasimha. Distributed synchronizers. In *ICPP '88*, pages 23–37, 1988.

[116] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In Giacobazzi [77], pages 246–264.

[117] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In R. L. Rudell, editor, *ICCAD 1995*, pages 2–6. IEEE Computer Society Press, 1995.

[118] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of Prolog programs. In J. H. Siekmann, editor, *8th CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 281–299. Springer, 1986.

[119] T. Kanamori and H. Seki. Verification of Prolog programs using an extension of execution. In E. Y. Shapiro, editor, *3rd ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 1986.

[120] J. Kelly. *The Essence of Logic*. Essence of Computing. Prentice Hall Europe, 1997.

[121] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th POPL*, pages 196–205. ACM Press, 1993.

[122] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[123] G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In A. Evans, S. Kent, and B. Selic, editors, *3rd UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 528–540. Springer, 2000.

[124] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *33rd POPL*, pages 115–126. ACM Press, 2006.

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

[126] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioral subset of UML Statechart diagrams using the SPIN model checker. *Formal Aspects of Computing*, 11(6):637–664, December 1999.

[127] M. Leuschel and T. Massart. Infinite-state model checking by abstract interpretation and program specialization. In Bossi [21], pages 62–81.

[128] J. Lilius and H. Sara. An implementation of UML state machine semantics for model checking. In *NWPT 1999*, 1999. URL http://www.docs.uu.se/nwpt99/proceedings/.

[129] J. Lind-Nielsen, H. R. Andersen, G. Behrmann, H. Hulgaard, K. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. In B. Steffen, editor, *4th TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 1998.

[130] R. J. Lipton. Reduction: A method for proving properties of parallel programs. In *2nd POPL*, pages 78–86. ACM Press, 1975. Communications of the ACM 18(12).

[131] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *8th FSE*, pages 120–129. ACM Press, 2000.

[132] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/w cet/benchmarks.html`.

[133] G. S. Manku. Structural symmetries and model checking. Master's thesis, Department of Computer Science and Engineering, University of California at Berkeley, 1997.

[134] G. S. Manku, R. Hojati, and R. Brayton. Structural symmetry and model checking. In Hu and Vardi [104], pages 159–171.

[135] Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8):491–502, August 1973.

[136] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[137] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[138] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[139] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *IFIP Congress 1962*. North-Holland, 1983.

[140] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In Etessami and Rajamani [64], pages 476–490.

[141] F. Mesnard, S. Hoarau, and A. Maillard. CLP($\mathcal{X}$) for automatically proving program properties. In F. Baader and K. U. Schulz, editors, *1st FroCoS*, volume 3 of *Applied Logic Series*. Kluwer Academic Publishers, 1996.

[142] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for Statecharts. In R. K. Shyamasundar and K. Ueda, editors, *3rd ASIAN*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 1997.

[143] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[144] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Monographs in Computer Science. Springer, 2001.

[145] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In PLDI2001 [159], pages 221–231. SIGPLAN Notices 36(5).

[146] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[147] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape, size and bag properties via separation logic. In B. Cook and A. Podelski, editors, *8th VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.

[148] R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Challenges in satisfiability modulo theories. In F. Baader, editor, *18th RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[149] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *1st CL*, volume 1861 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2000.

[150] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, March 2003. Version 1.5 formal/03-03-01.

[151] M. Pandey and R. K. Bryant. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. In Grumberg [82], pages 244–255.

[152] D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 59–78. Edinburgh University Press, 1969.

[153] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *6th CAV*, number 818 in Lecture Notes in Computer Science, pages 377–390. Springer, 1994.

[154] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996. Preliminary version appeared in 6th CAV [153].

[155] D. Peled. Ten years of partial order reduction. In Hu and Vardi [104], pages 17–28.

[156] G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient model checking of real-time systems using tabled logic programming and constraints. In Stuckey [188], pages 100–114.

[157] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2–3):197–230, 1999.

[158] A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existensial variables. In S. Etalle and M. Truszczynski, editors, *22nd ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2006.

[159] *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20–22, 2001*. ACM Press, May 2001. SIGPLAN Notices 36(5).

[160] A. Pnueli. Lecture 4: Deductive verification of CTL*. URL www.cs.nyu.edu/courses/spring04/G22.3033-017/lecture4.ps, 2004. Lecture Slides of *Advanced Topics in Reactive Verification*, NYU, Spring 2004.

[161] *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, January 16–18, 2002*. ACM Press, 2002. SIGPLAN Notices 37(1).

[162] *Annual Symposium on Principles of Programming Languages: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California, January 12–14, 2005*. ACM Press, 2005.

[163] I. Porres. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, Department of Computer Science, Åbo Akademi University, November 2001.

[164] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In W. Pugh and C. Chambers, editors, *18th PLDI*, pages 14–24. ACM Press, 2004.

[165] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In Grumberg [82], pages 143–154.

[166] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *17th LICS*, pages 55–74. IEEE Computer Society Press, 2002.

[167] R. Richey. *Adaptive Differential Pulse Code Modulation Using PICmicro Microcontrollers*. Microchip Technology, Inc., 1997.

[168] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In POPL2005 [162], pages 296–309.

[169] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In C. Hankin and I. Siveroni, editors, *12th SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2005.

[170] A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):464–509, 2004.

[171] A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program t4ransformations. In Graf and Schwartzbach [81], pages 172–187.

[172] A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In G. Berry, H. Comon, and A. Finkel, editors, *13th CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 25–37. Springer, 2001.

[173] R. Rugina. Quantitative shape analysis. In Giacobazzi [77], pages 228–245.

[174] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[175] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. *The XSB System Version 2.5 Volume 1: Programmer's Manual*, June 2003.

[176] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. *The XSB System Version 2.5 Volume 2: Libraries, Interfaces and Packages*, June 2003.

[177] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In *1st SoftMC*. Elsevier Science, 2001. Electronic Notes in Theoretical Computer Science 55(3).

[178] F. B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. Springer, 1997.

[179] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[180] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *34th ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer, 2007.

[181] A. J. H. Simons. On the compositional properties of UML Statechart diagrams. In *3rd ROOM*, 2000.

[182] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 26(4):702–734, July 2004.

[183] A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, April 2000.

[184] A. P. Sistla, L. Miliades, and V. Gyuris. SMC: A symmetry-based model checker for verification of liveness properties. In Grumberg [82], pages 464–467.

[185] SNU real-time benchmarks. URL `http://archi.snu.ac.kr/realtime/benchmark/`. .

[186] Software errors cost U.S. economy $59.5 billion annually: NIST asseses technical needs of industry to improve software testing. URL http://www.nist.gov/public_affairs/releases/n02-10.htm, June 2002. NIST News Release.

[187] M. E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[188] P. J. Stuckey, editor. *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29–August 1, 2002, Proceedings*, volume 2401 of *Lecture Notes in Computer Science*. Springer, 2002.

[189] Y. Tanabe, T. Takai, and K. Takahashi. Verification tools using abstraction. *Computer Software*, 22(1):2–44, January 2005.

[190] D. Tang, S. Malik, A. Gupta, and C. N. Ip. Symmetry reduction in SAT-based model checking. In Etessami and Rajamani [64], pages 125–138.

[191] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, chapter 4, pages 133–191. Elsevier Science, 1990. Second printing 1998.

[192] L. Urbina. Analysis of hybrid systems CLP($\mathcal{R}$). In Freuder [73], pages 451–467.

[193] L. Urbina. The generalized railroad crossing: Its symbolic analysis in CLP($\mathcal{R}$). In Freuder [73], pages 565–567.

[194] A. C. Uselton and S. A. Smolka. Compositional semantics for Statecharts using labeled transition systems. In B. Jonsson and J. Parrow, editors, *5th CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1994.

[195] M. von der Beeck. A comparison of statechart variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *FTRTFT 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

[196] F. Wang. Efficient data structure for fully symbolic verification of real-time systems. In Graf and Schwartzbach [81], pages 157–171.

[197] F. Wang. Efficient verification of timed automata with BDD-like data structures. In Zuck et al. [202], pages 189–205.

[198] F. Wang and K. Schmidt. Symmetric symbolic safety analysis of concurrent software with pointer data structures. In D. Peled and M. Y. Vardi, editors, *22nd FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2002.

[199] H. Weyl. *Symmetry*. Princeton University Press, 1952. First Princeton Science Library Printing, 1989.

[200] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time computing systems by constraint-solving. In D. Hogrefe and S. Leue, editors, *7th FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 243–258. Chapman & Hall, 1995.

[201] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.

[202] L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors. *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New*

York, NY, USA, January 9–11, 2002, Proceedings, volume 2575 of *Lecture Notes in Computer Science*. Springer, 2003.

Appendix A

Additional Modeling Examples

A.1 Modeling Real-Time Synchronization

Await statement which is introduced in Chapter 3 is useful for *synchronization*. Another commonly encountered synchronization mechanism in concurrent settings is by *signals* or *interrupts*. The common mechanism of signaling is that a process first declares its interest in a signal specifiable by a signal identifier, and then it goes to sleep or perform other tasks. We say that the process *waits* for the signal. The signal can then be generated by another process when certain conditions are met or certain set of tasks are completed. The generation of the signal triggers the waiting process after a small amount of time, which then awakes from its sleep or terminates what it is currently executing, and starts to execute a prespecified piece of code (called *signal handler* or *interrupt service routine*).

Signaling can be thought as another level of abstraction above busy waiting, since it is actually implemented in digital computers as busy waiting for the raising of the signal. However, usually the busy waiting in signaling is implemented by a specialized hardware separated from the CPU such that no CPU time is lost executing a loop.

A.1.1 Waiting Time

There is a modeling issue with regard the **await** statement introduced in Section 3.3. This is because it may take an indefinite amount of time before it succeeds. An implementation of an **await** statement in real programming languages would have used operating system facilities to check from time to time whether the condition is satisfied or not, which is the so-called *busy waiting*. The checking is not necessarily periodic, but the operating system's scheduling mechanism would guarantee that the checking of the condition will eventually be performed. Assuming a dedicated operating system, it is possible to provide the range of time difference between one check with another.

Suppose that the condition of an await statement is periodically checked every ε time units, where $\varepsilon_l \leq \varepsilon \leq \varepsilon_h$, and we have a program with two processes, and Process 1 contains an **await** statement of the form

$$\langle l \rangle \textbf{await } (\mathit{boolexpr})$$

Then we translate the above statement into the following two CLP clauses:

$$
\begin{array}{l}
p(l_1, l_2, \tilde{X}, T_1', T_2) \texttt{ :- } \\
\quad \neg \mathit{boolexpr}\theta, T_1 \leq T_2, T_1 + \varepsilon_l \leq T_1' \leq T_1 + \varepsilon_h, p(l_1, l_2, \tilde{X}, T_1, T_2). \\
p(\mathit{next\_label}(l_1), l_2, \tilde{X}, T_1', T_2) \texttt{ :- } \\
\quad \mathit{boolexpr}\theta, T_1 \leq T_2, T_1 + \varepsilon_l \leq T_1' \leq T_1 + \varepsilon_h, p(l_1, l_2, \tilde{X}, T_1, T_2).
\end{array}
$$

The first clause is the case when the condition does not hold and the busy waiting loop has to iterate one more time, while the second clause is the case when the test succeeds and control advances to the statement at the next program point.

A.1.2 The Modeling

We now allow a concurrent program text to use three new statements:

kill (*PosInt*) *Variable* := *Expr*

signal (*PosInt*)

signal sleep (*PosInt*)

The syntactic element *PosInt* denotes an expression that evaluates to a positive integer, usually simply a constant.

To represent the state of a signal in the CLP model, we use a functor $\mathit{signal}(\mathit{Id}, S, R)$, where the arguments Id, S and R are signal identifier, signal status and number of processes waiting for the signal, respectively. The signal status is either *up* or *down*, denoting whether the signal is raised or not.

We now explain the CLP model of the three statements above. In our explanation, we assume a two-process concurrent program, each with its corresponding clock variable. This setting can be trivially generalized to more than two processes.

$\langle l_2 \rangle$ **kill** (id) x_k := *expr* is used to generate a signal, and atomically assign the expression *expr* to the variable x. Assuming that it is executed by Process 2, we translate the above statement into the following backward CLP clauses:

$$\begin{array}{l} p(L_1, m, signal(id, up, R), X_1, \ldots, X_k', \ldots, X_n, S_1, S_2, T_1, T_2') \text{ :- } \\ \quad p(L_1, l_2, signal(id, down, R), X_1, \ldots, X_k, \ldots, X_n, S_1, S_2, T_1, T_2), \\ \quad X_k' = expr\theta, T_2 + \varepsilon_l \leq T_2' \leq T_2 + \varepsilon_h. \\ p(L_1, next\_label(l_2), signal(id, down, 0), X_1, \ldots, X_n, S_1, S_2, T_1, T_2) \text{ :- } \\ \quad p(L_1, m, signal(id, up, 0), X_1, \ldots, X_n, S_1, S_2, T_1, T_2). \end{array}$$

The first clause models the rising of a signal. Notice the change from $signal(id, down, R)$ to $signal(id, up, R)$. This signal sending takes ε amount of time, where $\varepsilon_l \leq \varepsilon \leq \varepsilon_h$. The program point m denotes a *wait location* of Process 2, where its value is different from program points or other wait locations of Process 2. The variables S_1 denotes the signal id Process 1 is currently waiting on, similarly with S_2 for Process 2. In the above clauses, these are unchanged. T_1 and T_2 are both clock variables.

The basic idea is that when a signal is raised, all other processes that wait for the signal are notified, and each of them acknowledges the receiving of the signal by decrementing by one the number of waiting process (the third argument of *signal*). When the number of waiting processes reaches 0, the signal flag can be lowered, which is modeled by the second CLP clause above. In this way, all processes that are waiting for the signal must have been serviced before the process executing the **kill** statement proceeds to execute the next statement.

$\langle l_1 \rangle$ **signal** (id) is used to declare that the current process is waiting for a signal. It basically increments the third argument of *signal* functor. In our two-process program, we assume that Process 1 executes this statement. In the backward CLP model of the program, the statement is translated into two clauses, of which the first one is:

$$\begin{array}{l} p(next\_label(l_1), L_2, signal(id, down, R+1), X_1, \ldots, X_n, S_1', S_2, T_1', T_2) \text{ :- } \\ \quad p(l_1, L_2, signal(id, down, R), X_1, \ldots, X_n, S_1, S_2, T_1, T_2), \\ \quad S_1' = id, T_1 + \varepsilon_l \leq T_1' \leq T_1 + \varepsilon_h. \end{array}$$

Notice that we set S_1 to the signal id, denoting that Process 1 is now waiting for that signal. The execution of the statement also consumes some time, which is within $[\varepsilon_l, \varepsilon_h]$.

The second clause handles the case when the signal is raised and Process 1 must jump to a signal handler.

$$\begin{aligned}
&p(m, L_2, signal(id, up, R), X_1, \ldots, X_n, S_1', S_2, T_1', T_2) \text{ :- } \\
&\quad p(L_1, L_2, signal(id, up, R+1), X_1, \ldots, X_n, S_1, S_2, T_1, T_2), \\
&\quad S_1' = 0, T_1 + \varepsilon_l \le T_1' \le T_1 + \varepsilon_h.
\end{aligned}$$

In the above clause, m is the start program point of the signal handler and L_1 is a variable denoting any program point of Process 1. The clause models the decrement of the third argument of *signal* functor, to declare that the process has been serviced. Moreover, since Process 1 should no longer wait for signal Id, it sets the value of S_1 to 0, to declare that Process 1 now waits for no signal (recall that a signal id is always a positive integer).

$\langle l_1 \rangle$ **signal sleep** (id) is used to declare that Process 1 waits for a signal id, and then it immediately goes to sleep, waiting for the signal to be raised. It is translated into the following two backward CLP clauses:

$$\begin{aligned}
&p(m, L_2, signal(id, down, R+1), X_1, \ldots, X_n, T_1, T_2) \text{ :- } \\
&\quad p(l_1, L_2, signal(id, down, R), X_1, \ldots, X_n, T_1, T_2). \\
&p(next\_label(l_1), L_2, signal(id, up, R), X_1, \ldots, X_n, T_1', T_2) \text{ :- } \\
&\quad p(m, L_2, signal(id, up, R+1), X_1, \ldots, X_n, T_1, T_2), \\
&\quad T_2 + \delta_l \le T_1' \le T_2 + \delta_h, T_1 + \varepsilon_l \le T_1'.
\end{aligned}$$

The first clause models the registering of Process 1 waiting for the signal id, while signal id is still down. Here we increase R by 1 in the tuple to denote that one more process is waiting for signal id. With this Process 1 also moves to program point m. We assume that the value of m is two more than the maximum program point value in the program text.

The second clause models the awakening of the sleeping Process 1 due to the rising of the corresponding signal by Process 2. Here, while signal id is up, we reduce the number of processes that is waiting for the signal from $R+1$ to R. An important thing here is that Process 1 would have waited for indefinite amount of time, when the signal is raised. Therefore the correct clock value when the Process 1 awakes must be the same as the clock value of Process 2. But here we add some δ delays in the execution, where $\delta_l \le \delta \le \delta_h$. Moreover, there is a minimum amount of time spent in sleep, at least for the execution time of the statement itself, which is ε_l.

We now show the flexibility of the above language constructs in modeling various synchro-

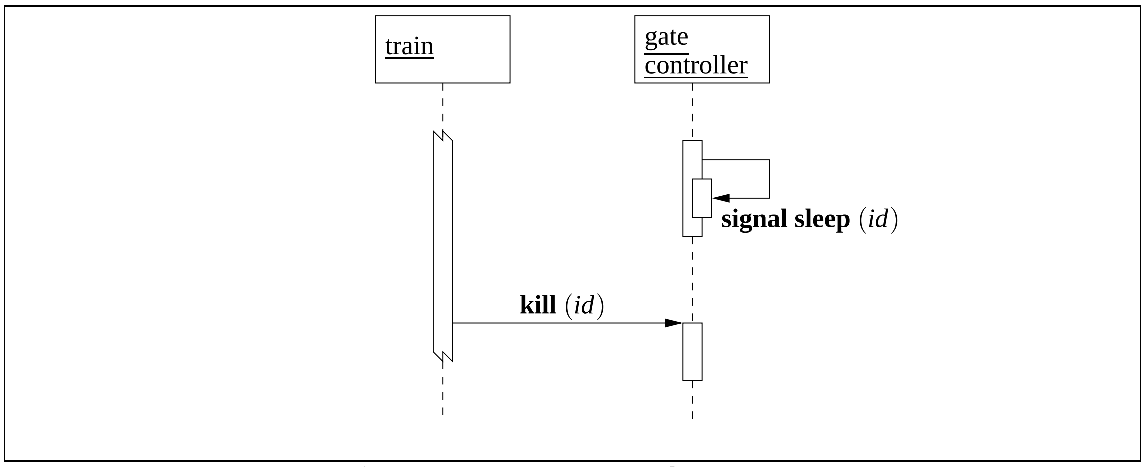

Figure A.1: One-Way Synchronization

nization paradigms found in the literature.

One-Way Synchronization. This mode of synchronization is useful to model an environment acting on a system. Such modeling of environment is adopted in Esterel synchronous language [87], where the environment “drives” the system by emitting signals one-way. An example of a train (environment in this case) sending a synchronous one-way message to a railway gate controller (the system) is shown using UML sequence diagram in Figure A.1. Note that solid arrows as in UML sequence diagrams denote a synchronous message. The self-call annotated with “**signal sleep** (id)” represents the registering of the interest in the signal by the gate controller. This is the activity modeled by the first CLP clause of the CLP modeling of **signal sleep** . The train may then trigger the signal, say when it is approaching the gate. This is the sending of the message annotated with **kill** (id). Correspondingly, the gate controller accepts the message and starts to lower the crossing gate. The message acceptance by the gate controller is modeled by the second clause of the CLP model of **signal sleep** (id).

Time-Triggered Protocol (TTP). TTP is used to implement *Time-Triggered Architecture* (*TTA*) [122]. TTA is becoming the standard architecture for modern embedded real-time systems due to its high predictability obtained through a global periodic clock[1].

Time-triggered architecture was proposed by Kopetz [122] to increase the timing predictability of distributed real-time systems. It is basically a middleware for distributed real-time systems, which takes advantage of a priori known information to ensure hard real-time constraints. As we have mentioned, TTA is based on a communication protocol called the Time-Triggered Protocol (TTP). The TTP is basically a Time-Division Multiple Access (TDMA) protocol which transmits

[1]Note that TTA/TTP does not actually require that the time difference between synchronization points to be periodic [122]. The important thing is that a synchronization will eventually occur after some time.

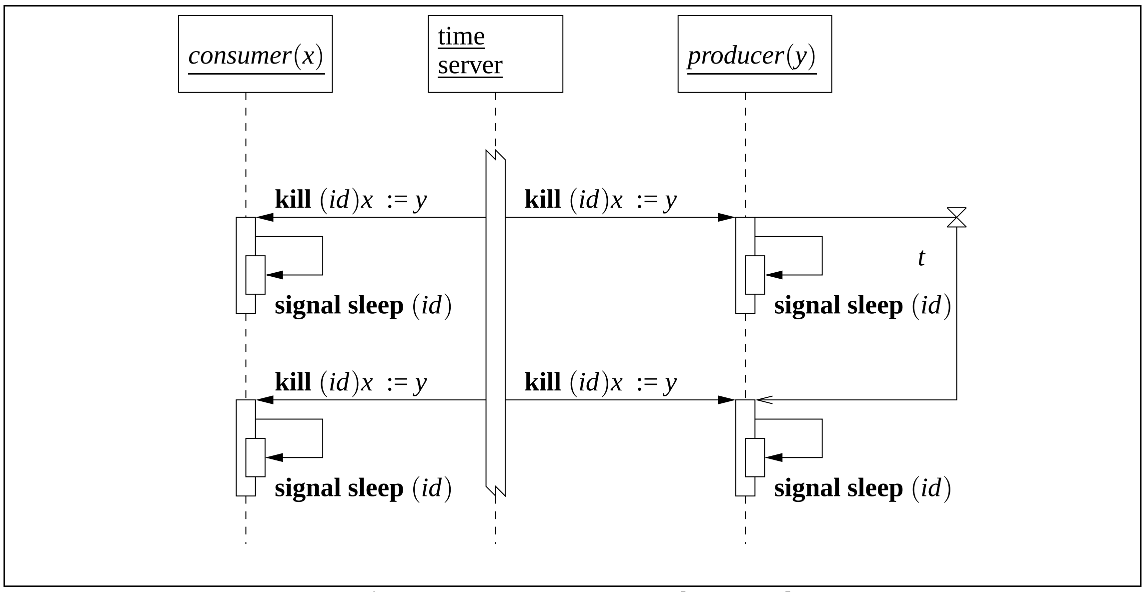

Figure A.2: Time-Triggered Protocol

messages from a node (time server) within some fixed time slice in a round-robin fashion.

Figure A.2 is a UML sequence chart depicting one full period of a time-triggered protocol communication. In such protocol, we always have a *time server,* which broadcasts a signal at every specified period t. In addition to a time server, in Figure A.2 there are also a producer and consumer which capture the signal. The producer produces a value which is stored in the variable y, while the consumer uses the value given as the variable x. In TTP, the exchange of data only happens at period boundary, and they occur "instantaneously." To model this, the atomic assignment in **kill** statement is handy. In Figure A.2, whenever the signal is risen, the y is assigned to x, modeling an instantaneous data transfer from the producer to the consumer. The waiting until the period expires at the time server can be implemented using a delay statement.

Execution steps of synchronous languages such as Esterel [15] can also be modeled in a similar manner. We can see the analogy of TTP with Esterel's semantics where the effect of a signal is only noticed within time region. Essentially TTA can be used as an implementation platform for synchronous languages such as Esterel.

Symmetric Synchronization (Barrier). *Synchronization barrier* (see e.g., [115, 72]) is a well-known technique to synchronize a number of parallel processes. According to [72], "A *barrier* is a particular point in a distributed computation that every process in a group must reach before any process can proceed further."

Barriers can be implemented using the constructs we have introduced so far. The basic mechanism is exemplified by the sequence diagram shown in Figure A.3. Two processes, Process 1 and 2 that are to be synchronized at some point have to reach certain stages of computation. We

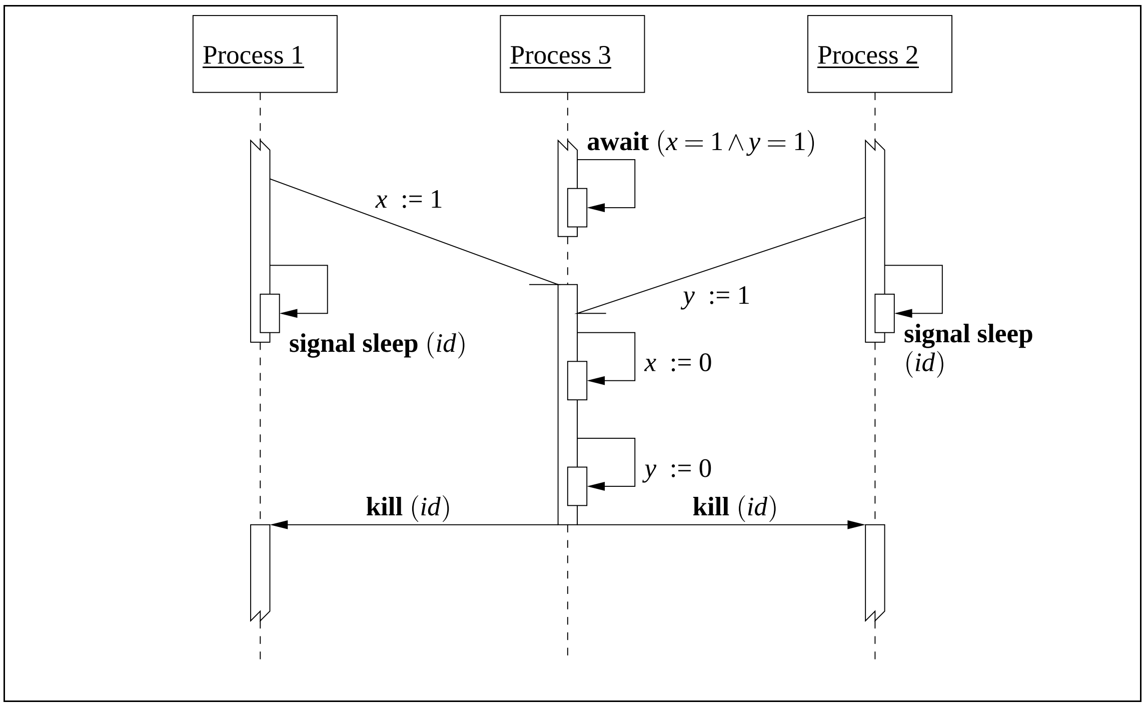

Figure A.3: Symmetric Synchronization (Barrier)

assume that when Process 1 has reached the stage it assigns x to 1. Similarly, Process 2 would assign y to 1. Here we assume that initially the values of x and y are both 0. After the assignments, both processes goes to sleep by executing **signal sleep** (id).

Some time after the assignments, Process 3, which have previously executed **await** $(x = 1 \wedge y = 1)$, becomes aware that Process 1 and 2 have set the values of x and y respectively to 1. This detection is depicted in Figure A.3 by the two sloped half-arrows, which denote *asynchronous* communication. Process 3 then resets both x and then y to 0. We assume that enough time has passed such that Process 1 and 2 are already asleep waiting for the signal (perhaps by introducing sufficient delay). At this point both Process 1 and 2 have reached the barrier. Process 3 then raises the *id* signal, which at the same time awakens both Process 1 and 2 to continue their executions.

Appendix B

Additional Proof Examples

B.1 Complete Proof of List Reverse Program

The assertion that we prove here is the following:

$$p(0,H,I,J,H_f,J_f),\mathit{alist}(H,I),J=0 \models \mathit{reverse}(H,I,0,H_f,J_f),\mathit{alist}(H_f,J_f).$$

The main proof is shown in Section B.1.1.

The proof requires an introduction of loop invariant in order to find a recursion in the unfolding of the loop. We therefore generalize the lhs of obligation $1'$ using the CUT rule into obligation 2. The use of CUT requires us to prove

$$\begin{aligned}&p(0,H,I,J,H_f,J_f),\mathit{alist}(H,I),H_0=H,I_0=I,J=0\\&\quad\models p(0,H,I,J,H_f,J_f),\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I).\end{aligned}$$

((5.14) on Page 151). The proof is shown in Section B.1.2.

From obligation 2 further proof will branch into two obligations. One of the branch 3a denotes the program's execution path that exits the loop, while the branch 3b denotes the path that enters the loop body, and eventually reaches $\langle 0\rangle$ again at assertion 6, which is proved by coinduction (AP application). The application of AP at 6 requires the proof of the side condition. The subsumption test is obligation 7s.1 and the residual obligation is 7r.1.

We assume that 7s.1 is directly proved (via DP), however, we are obliged to establish the following assertions:

A. $\mathit{reverse}(H_0, I_0, I, H, J), \mathit{alist}(H, J), \mathit{alist}(H, I), \mathit{no\_share}(H, J, I),$

$I \neq 0, H' = \langle H, I+1, J \rangle, I' = H[I+1], J' = I \models \mathit{reverse}(H_0, I_0, I', H', J').$

B. $\mathit{reverse}(H_0, I_0, I, H, J), \mathit{alist}(H, J), \mathit{alist}(H, I), \mathit{no\_share}(H, J, I),$

$I \neq 0, H' = \langle H, I+1, J \rangle, I' = H[I+1], J' = I \models \mathit{alist}(H', J').$

C. $\mathit{reverse}(H_0, I_0, I, H, J), \mathit{alist}(H, J), \mathit{alist}(H, I), \mathit{no\_share}(H, J, I),$

$I \neq 0, H' = \langle H, I+1, J \rangle, I' = H[I+1], J' = I \models \mathit{alist}(H', I').$

D. $\mathit{reverse}(H_0, I_0, I, H, J), \mathit{alist}(H, J), \mathit{alist}(H, I), \mathit{no\_share}(H, J, I),$

$I \neq 0, H' = \langle H, I+1, J \rangle, I' = H[I+1], J' = I \models \mathit{no\_share}(H', J', I').$

The proofs of A, B, C, and D are given in Sections B.1.3, B.1.4, B.1.5, and B.1.6, respectively.

The proof of B also requires that E be established (Section B.1.7). The proof of D requires that F is established (proof in Section B.1.8), which in turn requires that G (proof in Section B.1.9) is established.

The proofs of B, C, and D uses the separation principle (SEP) discussed in Section 5.9.1.

B.1.1 Main Proof of Linked List Reverse

1 $p(0,H,I,J,H_f,J_f), alist(H,I), J=0 \models reverse(H,I,0,H_f,J_f), alist(H_f,J_f)$

1′ $p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$
$\models reverse(H_0,I_0,H_f,J_f), alist(H_f,J_f)$ Simplified 1′

2 $p(0,H,I,J,H_f,J_f), reverse(H_0,I_0,H,J), alist(H,J), alist(H,I), no\_share(H,J,I)$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ CUT 1′

3a $p(\Omega,H,I,J,H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I),$
$no\_share(H,J,I), I=0$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ LU 2

3b $p(1,H,I,J,H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I),$
$no\_share(H,J,I), I\neq 0$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ LU 2

4 $reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f), alist(H_f,0), no\_share(H_f,J_f,0)$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ LU 3a

5 $\neg\Box$ DP 4

6 $p(0,H',I',J',H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I), no\_share(H,J,I)$
$I\neq 0, H'=\langle H,I+1,J\rangle, I'=H[I+1], J'=I$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ LU 3b

7 $\neg\Box$ AP 2,6

7s.1 $p(0,H',I',J',H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I), no\_share(H,J,I)$
$I\neq 0, H'=\langle H,I+1,J\rangle, I'=H[I+1], J'=I$
$\models p(0,H',I',J',H_f,J_f), reverse(H_0,I_0,H',J'), alist(H',J'),$
$alist(H',I'), no\_share(H',J',I')$ AP 2,6

7s.2 $\neg\Box$ See A,B,C,D

7r.1 $reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I),$
$no\_share(H,J,I), I\neq 0, H'=\langle H,I+1,J\rangle, I'=H[I+1], J'=I$
$\models reverse(H_0,I_0,0,H_f,J_f), alist(H_f,J_f)$ AP 2,6

7r.2 $\neg\Box$ DP 7r.1

B.1.2 Proof of CUT Side Condition for Linked List Reverse

2.1 $p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$
$\models p(0,H,I,J,H_f,J_f), reverse(H_0,I_0,I,H,J), alist(H,J), alist(H,I),$
$no\_share(H,J,I)$

2.2 $p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$
$\models p(0,H,I,J,H_f,J_f), alist(H,J), alist(H,I), no\_share(H,J,I),$
$H_0=H, I_0=I, J=0$ RU 2.1

2.3 $p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$
$\models p(0,H,I,J,H_f,J_f), alist(H,I), no\_share(H,J,I), H_0=H, I_0=I, J=0$ RU 2.2

2.4 $p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$
$\models p(0,H,I,J,H_f,J_f), alist(H,I), H_0=H, I_0=I, J=0$ RU 2.3

2.5 $\neg\Box$ DP 2.4

B.1.3 Proof of Assertion A

A.1	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),I\neq 0$	
	$H'=\langle H,I+1,J\rangle,I'=H[I+1],J'=I$	
	$\models \mathit{reverse}(H_0,I_0,I',H',J')$.	
A.1′	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),I\neq 0$	
	$\models \mathit{reverse}(H_0,I_0,H[I+1],\langle H,I+1,J\rangle,I)$.	Simplified A.1′
A.2	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),I\neq 0$	
	$\models \mathit{reverse}(H_0,I_0,I,H,J)$	RU A.1′
A.3	$\neg\Box$	DP A.2

B.1.4 Proof of Assertion B

B.1	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),I\neq 0,$	
	$H'=\langle H,I+1,J\rangle,I'=H[I+1],J'=I\models \mathit{alist}(H',J')$	
B.1′	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),I\neq 0$	
	$\models \mathit{alist}(\langle H,I+1,J\rangle,I)$	Simplified B.1′
B.2a	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),I\neq 0,J=0$	
	$\models \mathit{alist}(\langle H,I+1,J\rangle,I)$	LU B.1′
B.2b	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),$	
	$\mathit{no\_reach}(H,J,I),\mathit{no\_share}(H,H[J+1],I),I\neq 0,J\neq 0$	
	$\models \mathit{alist}(\langle H,I+1,J\rangle,I)$	LU B.1′
B.3	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),I\neq 0,J=0$	
	$\models I\neq 0,\mathit{alist}(\langle H,I+1,J\rangle,\langle H,I+1,J\rangle[I+1]),$	
	$\mathit{no\_reach}(\langle H,I+1,J\rangle,I,\langle H,I+1,J\rangle[I+1])$	RU B.2a
B.4	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),I\neq 0,J=0$	
	$\models I\neq 0,\mathit{alist}(\langle H,I+1,J\rangle,J),\mathit{no\_reach}(\langle H,I+1,J\rangle,I,J)$	AIP B.3
B.5	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),I\neq 0,J=0$	
	$\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,J),I\neq 0,J=0$	RU B.4
B.6	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),I\neq 0,J=0$	
	$\models I\neq 0,J=0$	RU B.5
B.7	$\neg\Box$	DP B.6
B.8	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),$	
	$\mathit{no\_reach}(H,J,I),\mathit{no\_share}(H,H[J+1],I),I\neq 0,J\neq 0$	
	$\models \mathit{alist}(\langle H,I+1,J\rangle,\langle H,I+1,J\rangle[I+1]),$	
	$\mathit{no\_reach}(\langle H,I+1,J\rangle,I,\langle H,I+1,J\rangle[I+1])$	RU B.2b
B.9	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),$	
	$\mathit{no\_reach}(H,J,I),\mathit{no\_share}(H,H[J+1],I),I\neq 0,J\neq 0$	
	$\models \mathit{alist}(\langle H,I+1,J\rangle,J),\mathit{no\_reach}(\langle H,I+1,J\rangle,I,J)$	AIP B.8
B.10	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),$	
	$\mathit{no\_reach}(H,J,I),\mathit{no\_share}(H,H[J+1],I),I\neq 0,J\neq 0$	
	$\models \mathit{alist}(H,J),\mathit{no\_reach}(\langle H,I+1,J\rangle,I,J)$	SEP B.9
B.11	$\neg\Box$	DP B.10 with E.1

B.1.5 Proof of Assertion C

C.1	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I)$	
	$I \neq 0, H' = \langle H, I+1, J\rangle, I' = H[I+1], J' = I \models \mathit{alist}(H',I')$	
C.1′	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{alist}(\langle H, I+1, J\rangle, H[I+1])$	Simplified C.1
C.2	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0 \models \mathit{alist}(\langle H, I+1, J\rangle, H[I+1])$	LU C.1′
C.3	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0 \models \mathit{alist}(H, H[I+1])$	SEP C.2
C.4	$\neg\Box$	DP C.3

B.1.6 Proof of Assertion D

D.1	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I),$	
	$I \neq 0, H' = \langle H, I+1, J\rangle, I' = H[I+1], J' = I$	
	$\models \mathit{no\_share}(H',J',I')$	
D.1′	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,I),\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{no\_share}(\langle H, I+1, J\rangle, I, H[I+1])$	Simplified D.1
D.2	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{no\_share}(\langle H, I+1, J\rangle, I, H[I+1])$	LU D.1′
D.3	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{no\_reach}(\langle H, I+1, J\rangle, I, H[I+1]),$	
	$\mathit{no\_share}(\langle H, I+1, J\rangle, \langle H, I+1, J\rangle[I+1], H[I+1]), I \neq 0$	RU D.2
D.4	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{no\_reach}(\langle H, I+1, J\rangle, I, H[I+1]),$	
	$\mathit{no\_share}(\langle H, I+1, J\rangle, J, H[I+1]), I \neq 0$	AIP D.3
D.5	$\mathit{reverse}(H_0,I_0,I,H,J),\mathit{alist}(H,J),\mathit{alist}(H,H[I+1]),\mathit{no\_reach}(H,I,H[I+1]),$	
	$\mathit{no\_share}(H,J,I), I \neq 0$	
	$\models \mathit{no\_reach}(\langle H, I+1, J\rangle, I, H[I+1]),$	
	$\mathit{no\_share}(H, J, H[I+1]), I \neq 0$	SEP D.4
D.6	$\neg\Box$	DP D.5 with F.1

B.1.7 Proof of Assertion E

E.1	$\mathit{no\_reach}(H,J,I), \mathit{no\_share}(H,H[X+1],I), X \neq 0, I \neq 0$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	
E.2a	$\mathit{no\_reach}(H,J,I), H[X+1]=0, X \neq 0, I \neq 0$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	LU E.1
E.2b	$\mathit{no\_reach}(H,J,I), \mathit{no\_reach}(H,H[X+1],I), \mathit{no\_share}(H,H[H[X+1]+1],I),$ $X \neq 0, I \neq 0, H[X+1] \neq 0$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	LU E.1
E.3	$\mathit{no\_reach}(H,J,I), X \neq 0, I \neq 0, H[X+1]=0$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]), X \neq 0$	RU E.2a
E.4	$\mathit{no\_reach}(H,J,I), X \neq 0, I \neq 0, H[X+1]=0$ $\models X \neq 0, H[X+1]=0$	RU E.3
E.5	$\neg\Box$	DP E.4
E.6	$\neg\Box$	AP E.1,E.2b

E.6s.1	$\mathit{no\_reach}(H,J,I), \mathit{no\_reach}(H,H[X+1],I), \mathit{no\_share}(H,H[H[X+1]+1],I),$ $X \neq 0, I \neq 0, H[X+1] \neq 0$ $\models \mathit{no\_reach}(H,J,I), \mathit{no\_share}(H,H[H[X+1]+1],I),$ $H[X+1] \neq 0, I \neq 0$	AP E.1,E.2b
E.6s.2	$\neg\Box$	DP E.6s.1

E.6r.1	$\mathit{no\_reach}(H,J,I), \mathit{no\_reach}(H,H[X+1],I), \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]),$ $X \neq 0, I \neq 0, H[X+1] \neq 0$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	AP E.1,E.2b
E.6r.2	$\mathit{no\_reach}(H,J,I), \mathit{no\_reach}(H,H[X+1],I), \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]),$ $X \neq 0, I \neq 0, H[X+1] \neq 0$ $\models X \neq 0, \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1])$	RU E.6r.1
E.6r.3	$\neg\Box$	DP E.6r.2

B.1.8 Proof of Assertion F

F.1	$\mathit{no\_share}(H,J,I),\mathit{no\_reach}(H,I,H[I+1]),I\neq 0 \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_share}(H,J,H[I+1]),I\neq 0$	
F.2a	$\mathit{no\_reach}(H,I,H[I+1]),I\neq 0,J=0 \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_share}(H,J,H[I+1]), I\neq 0$	LU F.1
F.2b	$\mathit{no\_reach}(H,J,I),\mathit{no\_share}(H,H[J+1],I),\mathit{no\_reach}(H,I,H[I+1]),I\neq 0,J\neq 0 \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_share}(H,J,H[I+1]), I\neq 0$	LU F.1
F.3	$\mathit{no\_reach}(H,I,H[I+1]),I\neq 0,J=0 \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),I\neq 0,J=0$	RU F.2a
F.4	$\neg\Box$	DP F.3 with G.1
F.5	$\mathit{no\_reach}(H,J,H[I+1]),\mathit{no\_share}(H,H[J+1],I),\mathit{no\_reach}(H,I,H[I+1]), I\neq 0,J\neq 0,I\neq J \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_share}(H,J,H[I+1]), I\neq 0$	LU F.2b
F.6	$\neg\Box$	AP F.1,F.5

F.6s.1	$\mathit{no\_reach}(H,J,H[I+1]),\mathit{no\_share}(H,H[J+1],I),\mathit{no\_reach}(H,I,H[I+1]), I\neq 0,J\neq 0,I\neq J \models \mathit{no\_share}(H,H[J+1],I),\mathit{no\_reach}(H,I,H[I+1]),I\neq 0$	AP F.1,F.5
F.6s.2	$\neg\Box$	DP F.6s.1

F.6r.1	$\mathit{no\_reach}(H,J,H[I+1]),\mathit{no\_reach}(\langle H,I+1,H[J+1]\rangle,I,H[I+1]), \mathit{no\_reach}(H,I,H[I+1]),\mathit{no\_share}(H,H[J+1],H[I+1]),I\neq 0,J\neq 0,I\neq J \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_share}(H,J,H[I+1]), I\neq 0$	AP F.1,F.5
F.6r.2	$\mathit{no\_reach}(H,J,H[I+1]),\mathit{no\_reach}(\langle H,I+1,H[J+1]\rangle,I,H[I+1]), \mathit{no\_reach}(H,I,H[I+1]),\mathit{no\_share}(H,H[J+1],H[I+1]),I\neq 0,J\neq 0,I\neq J \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[I+1]),\mathit{no\_reach}(H,J,H[I+1]), \mathit{no\_share}(H,H[J+1],H[I+1]),I\neq 0,J\neq 0$	RU F.6r.1
F.6r.2	$\neg\Box$	DP F.6r.2 with G.1

B.1.9 Proof of Assertion G

G.1	$\mathit{no\_reach}(H,I,X) \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	
G.2a	$X=0 \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	LU G.1
G.2b	$\mathit{no\_reach}(H,I,H[X+1]),X\neq 0,I\neq X \models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	LU G.1
G.3	$X=0 \models X=0$	RU G.2a
G.4	$\neg\Box$	DP G.3
G.5	$\neg\Box$	AP G.1,G.2b

G.5s.1	$\mathit{no\_reach}(H,I,H[X+1]),X\neq 0,I\neq X \models \mathit{no\_reach}(H,I,H[X+1])$	AP G.1,G.2b
G.5s.2	$\neg\Box$	DP G.5s.1

G.5r.1	$\mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]),X\neq 0,I\neq X$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,X)$	AP G.1,G.2b
G.5r.2	$\mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]),X\neq 0,I\neq X$ $\models \mathit{no\_reach}(\langle H,I+1,J\rangle,I,H[X+1]),X\neq 0,I\neq X$	RU G.5r.1
G.5r.3	$\neg\Box$	DP G.5r.2