Dipartimento di Informatica e
Scienze dell'Informazione

A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

by

Marco Bozzano

Theses Series **DISI-TH-2002-01**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

by

Marco Bozzano

January, 2002

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Marco Bozzano
DISI, Univ. di Genova
bozzano@disi.unige.it

Date of submission: January 2002

Title: A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

Advisors: Maurizio Martelli
DISI, Univ. di Genova
martelli@disi.unige.it

Giorgio Delzanno
DISI, Univ. di Genova
giorgio@disi.unige.it

Ext. Reviewers: Jean-Marc Andreoli
Xerox Research Centre Europe, France
Jean-Marc.Andreoli@xrce.xerox.com

Maurizio Gabbrielli
Dipartimento di Scienze dell'Informazione
Univ. di Bologna, Italy
gabbri@cs.unibo.it

Frank Pfenning
Department of Computer Science
Carnegie Mellon University, U.S.A.
fp@cs.cmu.edu

Abstract

The aim of this thesis is to investigate the problem of verification for concurrent systems. In particular, a major problem in verification is that of validating systems, e.g. protocols, which are *parametric*, in the sense that the number of entities taking part in a given run is not fixed *a priori*. Typically, such kind of systems are also *infinite-state*, in that they use data structures containing possibly unbounded data values.

In this thesis we tackle the problem of verification using a logical approach. In particular, the leading thread of this work will be a specification language based on a fragment of Girard's linear logic, which we will show to have direct connections with classical formalisms like high-level nets or rewriting. By combining the power of logical connectives with the flexibility of rewriting, we are able to nicely model local and global transitions, and to elegantly express new data generation. Reasoning on heterogeneous domains can also be achieved via specialized constraint solvers.

We show how this language can be used both for the specification and the analysis of parametric systems. In particular, we present a verification procedure which resembles classical symbolic model checking algorithms for infinite-state systems, and is well-suited to study system properties like *safety*, e.g. *mutual exclusion*. Technically, our verification procedure uses a fixpoint computation strategy which is based on a new *bottom-up* semantics for a fragment of linear logic. We illustrate our methodology presenting different examples coming from concurrency theory, like a parameterized version of the *ticket* mutual-exclusion protocol, and from security, like authentication protocols.

To my family

Ilsa, I'm no good at being noble, but it doesn't take much to see that the problems of three little people don't amount to a hill of beans in this crazy world. Someday you'll understand that. Now, now . . . Here's looking at you, kid.

(H. Bogart as Rick Blaine in *Casablanca* (1942))

Acknowledgements

I am very grateful to the people that helped me during all these years. First of all, I want to thank my supervisor (and advisor), Maurizio Martelli, for his precious advice and support, and especially for his friendly guidance during the four years of my PhD program. I would also like to thank Eugenio Moggi (the coordinator of my PhD program), Giuseppe Rosolini and all my colleagues at DISI.

I wish to thank, in particular, Catherine Meadows and Frank Pfenning for giving me the opportunity to spend some months at the Naval Research Laboratory, in Washington D.C., and at Carnegie Mellon University, in Pittsburgh. Special thanks to Iliano Cervesato, Catherine Meadows and all the staff at NRL, including Paul Syverson, Ira Moskowitz, and Li Wu Chang. A special greeting to Matteo Pradella. I am grateful to Frank Pfenning for giving me the opportunity to visit one among the best computer science departments in the world, and for the several discussions I had with him. Special thanks and greetings to Jeff Polakow, David Walker, Andreas Abel, Brigitte Pientka, and also to the Italian community in Pittsburgh. I also wish to acknowledge some fruitful discussions I had with Jack Minker, Dale Miller and Catuscia Palamidessi.

I would also like to thank the reviewers of this thesis, Jean-Marc Andreoli, Maurizio Gabbrielli, and Frank Pfenning, for their helpful and pertinent comments and suggestions.

Finally, I wish to express my deepest gratitude to Giorgio Delzanno, who has also been advisor of this thesis, for his continuous and invaluable advice. Giorgio introduced me to the literature on verification and model checking, and shared with me the work of the last four years. I've been knowing Giorgio since the time I was an undergraduate and I really appreciated working with him. I must thank him for teaching me to be more confident in myself and a bit more enthusiastic in my daily work. It's probably not excessive to say that the present thesis would have never been conceivable without his help and support.

Table of Contents

List of Figures **6**

List of Tables **9**

Chapter 1 Introduction **10**

1.1 Motivations 10

1.2 Contribution of the Thesis 11

1.3 Plan of the Thesis 15

1.3.1 Structure of the Thesis 16

Chapter 2 Preliminaries **18**

2.1 Functions and Orders 18

2.2 Multisets 19

2.3 Signatures and Algebras 20

2.4 Substitutions and Multiset Unifiers 20

2.5 Basics of Logic Programming 21

I Linear Logic as a Unifying View of Concurrency **24**

Chapter 3 Linear Logic **25**

3.1 Full Linear Logic 25

3.1.1 Programming in Linear Logic 27

3.2 The Language LO 29
3.3 Extensions of LO 34
3.3.1 LO with **1** and $\otimes$ 34
3.3.2 LO with Universal Quantification 36

Chapter 4 LO, Multiset Rewriting and Petri Nets 40
4.1 Multiset Rewriting over Atomic Formulas 40
4.2 Place/Transition Nets 41
4.2.1 A Producer/Consumer Example 43
4.2.2 Place Invariants 44
4.3 From Petri Nets to Linear Logic 46
4.4 Petri Nets with Transfer Arcs 48
4.4.1 A Readers/Writers Example 49
4.4.2 Broadcast Protocols 50

Chapter 5 First-Order LO, First-Order Multiset Rewriting and Coloured Petri Nets 53
5.1 Multiset Rewriting Systems over First-Order Atomic Formulas 53
5.2 CP-nets 55
5.2.1 An Example: the Dining Philosophers 57
5.3 From CP-nets to Linear Logic 58
5.4 Multiset Rewriting with Universal Quantification 61

II Model Checking for Linear Logic Specifications 65

Chapter 6 A Backward Approach to Model Checking 66
6.1 Transition Systems and Verification 67
6.1.1 Structured Transition Systems 69
6.1.2 Symbolic Verification 73

6.2 Ensuring Termination 76
6.2.1 The Theory of Well-Quasi-Orderings 76
6.2.2 Well-Structured Transition Systems 78
6.3 Verification in Linear Logic 79
6.4 Related Work 80

Chapter 7 Bottom-Up Evaluation of Propositional LO Programs 82
7.1 A Bottom-Up Semantics for LO 83
7.2 An Effective Semantics for LO 88
7.3 Examples 96
7.3.1 A Producer/Consumer Example 97
7.3.2 A Petri Net for Mutual Exclusion 99
7.4 Related Work 102

Chapter 8 Refining LO Resource Management: LO with 1 104
8.1 A Bottom-Up Semantics for $\mathrm{LO}_{\mathbf{1}}$ 105
8.2 Constraint Semantics for $\mathrm{LO}_{\mathbf{1}}$ 108
8.3 Bottom-Up Evaluation for $\mathrm{LO}_{\mathbf{1}}$ 114
8.4 An Example: Readers/Writers 117
8.4.1 Verification of the Readers/Writers Protocol 120
8.5 Related Work 122

Chapter 9 Reasoning on Specialized Domains: LO with Constraints 124
9.1 Enriching LO With Constraints 125
9.1.1 Constraint Systems 125
9.1.2 The Language LO($\mathcal{C}$) 127
9.2 An Effective Semantics for LO($\mathcal{C}$) 129
9.3 Ensuring Termination 142
9.3.1 A Dynamic Abstraction from **DC** to **NC** 147

9.4 An Example: The Ticket Protocol 148
9.4.1 Specifying the Ticket Protocol 149
9.4.2 Verifying the Ticket Protocol 151
9.5 Related Work 154

Chapter 10 Bottom-Up Evaluation of First-Order LO Programs 157
10.1 A Proof-system for $LO_\forall$ 158
10.2 A Bottom-Up Semantics for $LO_\forall$ 161
10.3 An Effective Semantics for $LO_\forall$ 167
10.4 Ensuring Termination 181
10.5 An Example: A Distributed Test-and-Lock Protocol 183
10.6 Related Work 187

Chapter 11 A Case-Study: Security Protocols 189
11.1 Introduction 190
11.2 Some Background on Authentication 191
11.2.1 Cryptographic Prerequisites 192
11.2.2 A Classification of Attacks 193
11.2.3 The Dolev-Yao Intruder Model 193
11.2.4 An Informal Protocol Notation 194
11.3 Specifying Authentication Protocols 196
11.4 Examples 198
11.4.1 Millen's *ffgg* protocol 198
11.4.2 The Needham-Schroeder Protocol 202
11.4.3 Corrected Needham-Schroeder 205
11.4.4 The Otway-Rees Protocol 206
11.4.5 Experimental Results 209
11.5 Future Work 210

11.6 Related Work 211

Chapter 12 Conclusions **214**

12.1 Future Work 215

Bibliography **218**

Appendix A Verification Tool **233**

A.1 General Description 233

A.2 Implementation Notes 234

A.3 How It Works 234

A.4 Experimental Environment 236

A.5 Future Work 236

List of Figures

1.1 A simple Petri net representing a semaphore 12

3.1 A one-sided, dyadic proof system for full linear logic 27
3.2 A proof system for LO 32
3.3 Examples of LO proofs 33
3.4 A proof system for $LO_{\mathbf{1}}$ 34
3.5 Examples of $LO_{\mathbf{1}}$ proof 36
3.6 A proof system for $LO_{\forall}$ 37
3.7 An example of $LO_{\forall}$ proof 38

4.1 A producer/consumer net 43
4.2 Matrix and initial marking representation 45
4.3 Reachability as provability in propositional LO 48
4.4 A readers/writers net 49
4.5 A readers/writers broadcast protocol 51

5.1 The dining philosophers net 57
5.2 Reachability as provability in first-order LO 60
5.3 Reachability as provability in $LO_{\forall}$ 63

6.1 An abstract verification procedure for reachability 69
6.2 Illustrating the covering problem: a Petri net for mutual exclusion 70

6.3 A verification procedure for covering 73

7.1 A proof system for propositional LO 84
7.2 Symbolic fixpoint computation 94
7.3 Another example of LO derivation 95
7.4 Fixpoint computed for the producer/consumer example 97
7.5 A petri net for mutual exclusion: example trace 99
7.6 Fixpoint computed for the mutual exclusion example 100
7.7 Fixpoint computation for the mutual exclusion example: first step 101

8.1 A proof system for propositional $\mathrm{LO}_{\mathbf{1}}$ 105
8.2 Symbolic fixpoint computation for an $\mathrm{LO}_{\mathbf{1}}$ program 115
8.3 Invalidation phase for the readers/writers example 119
8.4 Fixpoint computed using invariant strengthening for the readers/writers protocol 121

9.1 A proof system for $\mathrm{LO}(\mathcal{C})$ 129
9.2 High-level description of the ticket protocol 149
9.3 A multi-client, single-server version of the ticket protocol 150
9.4 A multi-client, multi-server version of the ticket protocol 151
9.5 Multi-client ticket protocol: example trace 152

10.1 A proof system for non ground semantics of $\mathrm{LO}_{\forall}$ 160
10.2 A trace violating mutual exclusion for the test-and-lock protocol 184
10.3 A test-and-lock protocol: example trace 184
10.4 Fixpoint computed for the test-and-lock protocol 185
10.5 Fixpoint computed using invariant strengthening for the test-and-lock protocol 186

11.1 Specification of the *ffgg* protocol 199
11.2 Intruder theory for the *ffgg* protocol 200

11.3 A parallel session attack to the *ffgg* protocol 201
11.4 Specification of the Needham-Schroeder protocol 203
11.5 Intruder theory for the Needham-Schroeder protocol 204
11.6 An attack to the Needham-Schroeder protocol 204
11.7 Specification of the Otway-Rees protocol 207
11.8 Intruder theory for the Otway-Rees protocol 208
11.9 An attack to the Otway-Rees protocol 209

A.1 Pseudo-algorithm for bottom-up evaluation 235

List of Tables

3.1 The set of linear logic connectives 26

6.1 A parallel between reachability problems and linear logic semantics 79

9.1 Validating the ticket protocol: experimental results 153

11.1 Analysis of authentication protocols: experimental results 209

12.1 A summary of the experiments carried out in the thesis 216

Chapter 1

Introduction

The contribution of this thesis is twofold. First of all, we present a specification language, based on a fragment of Girard's linear logic, which is suitable for the specification of concurrent systems, and is closely related to well-known formalisms like Petri nets, high-level Petri nets or rewriting systems. The core of the language provides logical primitives to express, e.g., communication by *rendezvous*, *broadcast*, and fresh name generation, and can be extended with specialized constraint solvers to reason on different data domains. Furthermore, we enrich this language with a novel computational model based on an alternative *bottom-up* semantics for linear logic programs, which we show to have applications for the verification of *parameterized* and *infinite-state* systems.

We have obtained some original results, both theoretical and practical. From the theoretical side, we have isolated fragments of first order linear logic for which provability is decidable. From the practical side, we have derived an automated verification procedure for an interesting class of *safety* properties, and some novel results concerning the validation of parameterized systems.

1.1 Motivations

In recent years several attempts have been made in order to develop tools for the *automated* verification of concurrent and distributed systems. Specifically, one of the more challenging research areas is related to the verification of *parameterized* systems. By parameterized system, we mean any collection of an *arbitrary* but *finite* number of components interacting via synchronous or asynchronous communication. Examples of parameterized systems naturally arise in different areas of computer science like, e.g., operating systems, network computing, or web applications. For instance, communication and authentication protocols

are a key ingredient for applications involving remote access or e-commerce.

Very often, most of these applications are highly critical. They must comply with a number of security requirements, and therefore their design and implementation is always difficult and error-prone. As a meaningful example, we could mention the case of Needham-Schroeder authentication protocol [NS78], an apparently very simple protocol which only recently (*seventeen* years after its publication) has been shown to be flawed [Low95]. For these reasons, the availability of possibly automatic tools for the formal specification and analysis of such kind of systems seems to be indispensable.

Some interesting results have been obtained for the verification of the class of parameterized systems with *finite-state* components [ACJT96, FS01, PRZ01, APR+01]. In many practical cases, verification problems for this kind of systems can be reduced to problems on Petri nets, by applying a *counting* abstraction that simply forgets local data while keeping track of the number of processes in a given state. The original properties can then be verified on the resulting model by means of reachability procedures [GS92, EN98, EFM99, Del00]. However, very often such kind of systems are also *infinite-state*, in that they use data structures containing possibly *unbounded* data values. Some more results have been obtained for the verification of classes of infinite-state systems consisting of a *fixed* number of components with possibly *unbounded* data. As an example, in [BGP97, Del00, DP99, Fri99], *constraints* are used to symbolically represent and manipulate infinite collections of states. The new frontier seems therefore to be the verification of systems which are both *parametric* and *infinite-state*.

1.2 Contribution of the Thesis

In this thesis we will present a uniform logical environment for the specification and analysis of concurrent systems. In particular, we will isolate a fragment of Girard's linear logic [Gir87] corresponding to the language LO as defined in [AP91b] and some extensions of its. We will show that this language has close connections with classical formalisms in concurrency theory like (coloured) Petri nets or multiset rewriting systems over atomic formulas. Our work has in fact been inspired by the multiset-rewriting-like logic described in [CDL+99]. The core of the language provides logical primitives to express local communication and synchronization between processes, and to implement *broadcast* communication. Furthermore, universal quantification can be used as an elegant way to express fresh name generation.

Building upon previous works for verification of parameterized systems [AJ98, Del00, FS01, AJ01b], we will study a *backward* verification procedure, resembling usual symbolic model checking algorithms, which can be used to validate systems specified in our language. This procedure works by computing the *backward reachability set* of a given system starting

from a symbolic representation of the set states *violating* the property under consideration.

To exemplify, consider the simple Petri net [Rei85] drawn below, which represents a semaphore enforcing *mutual exclusion* for accessing an ideal (shared) resource. We have K

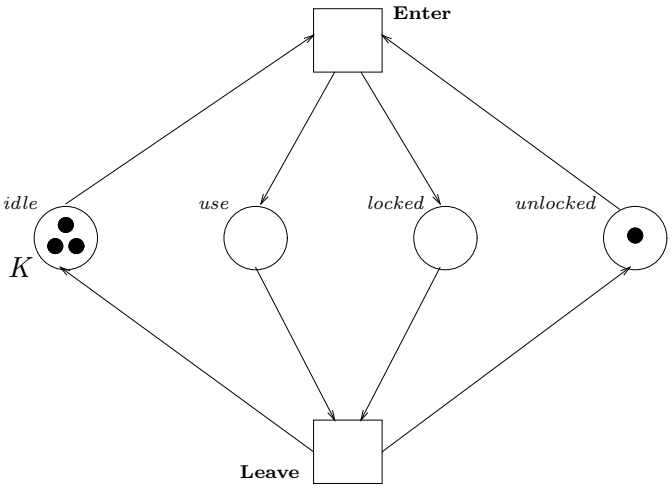

Figure 1.1: A simple Petri net representing a semaphore

processes (we have drawn three of them in Figure 1.1), represented by *tokens*, i.e., black spots, which can be in state *idle* or *use* (i.e., using the resource). The semaphore is in turn represented by a token which can assume one of the two values *locked* and *unlocked*. Initially *all* processes are idle and the semaphore is unlocked. A *configuration* of the system is a *multiset*, for instance the initial configuration drawn in Figure 1.1 is represented by the multiset $\{idle, idle, idle, unlocked\}$. A process can get control of the resource (transition Enter) by locking the semaphore, provided the semaphore is currently unlocked. A process can release the resource (transition Leave) by unlocking the semaphore. These transitions can be encoded in linear logic as the following two program *clauses* (let P be the resulting program):

$$\text{Enter:} \quad idle \parr unlocked \multimap use \parr locked$$
$$\text{Leave:} \quad use \parr locked \multimap idle \parr unlocked$$

The connective $\parr$ (multiplicative linear disjunction) acts as a multiset constructor, whereas $\multimap$ (reversed linear implication) can be viewed as a rewriting operator (i.e., the left-hand side can be rewritten into the right-hand side). For instance, the first clause can transform the initial configuration into $\{idle, idle, use, locked\}$.

Suppose we want to formally verify that the above property of mutual exclusion holds. Our verification strategy works as follows. The set of states *violating* the mutual exclusion property are exactly those in which there are *at least* two processes using the resource. Therefore we can *symbolically* represent them by the *multiset* $\{use, use\}$ (note that, in the

general case with K initially idle processes, this is a symbolic representation for an *infinite* set of system configurations). From a logical point of view, this is the counterpart of the linear logic axiom

$$\text{Unsafe:} \quad use \parr use \multimap \top$$

which, intuitively, states that *any* configuration containing *at least* two tokens use is provable, according to the following proof scheme

$$\dfrac{\dfrac{}{P \vdash \top, \Delta}\ \top_r}{P \vdash use, use, \Delta}\ \text{Unsafe}$$

where Δ is any multiset of tokens.

In order to verify if an unsafe configuration is *reachable*, we can compute the *predecessor* operator. If a process is in state use, necessarily it must have got access to the resource by executing the transition Enter, and the semaphore must have been unlocked (also, the semaphore must be currently locked). The corresponding set of predecessor states is given by $\{use, idle, unlocked\}$. The concept of *reachability* between Petri net configurations is the counterpart of the notion of *provability* in linear logic. For example, the following proof states that the configuration $\{use, idle, unlocked\}$ is *backward reachable* from $\{use, use, locked\}$:

$$\dfrac{\dfrac{\dfrac{}{P \vdash \top, locked}\ \top_r}{P \vdash use, use, locked}\ \text{Unsafe}}{P \vdash use, idle, unlocked}\ \text{Enter}$$

Note that the configuration $\{use, use, locked\}$ is *unsafe*, i.e., violates mutual exclusion. The above proof shows that the configuration $\{use, idle, unlocked\}$ is also unsafe.

Using a similar kind of reasoning, we can compute the reflexive and transitive closure of the predecessor operator, which yields the following (symbolic representations of) states:

$$\{use, use\} \qquad \{use, idle, unlocked\} \qquad \{idle, unlocked, idle, unlocked\}$$

None of the above states is a *legal initial configuration* of the system (remember that we require all processes to be initially idle, furthermore we allow only one *unlocked* token). In other words, any legal initial configuration will never lead to a configuration violating mutual exclusion. The property of mutual exclusion is therefore verified. From a logical point of view, computing the closure of the predecessor operator amounts to computing the *fixpoint* of a suitable operator.

This simple idea, illustrated for a specification given via a Petri net, in this thesis will be extended in order to reason about systems which can be *parametric* in *several* dimensions.

For instance, we can extend the above specification in order to allow more than one *resource*. An atomic formula $sem(x, locked)$ or $sem(x, unlocked)$ can be used to represent a semaphore for a resource named x. We can then verify mutual exclusion for *any* number of resources. We can also verify more complex protocols, which are *infinite-state* even for *fixed* values of the parameters. Our approach therefore provides a methodology to lift traditional verification techniques for Petri nets to more complex specification languages like coloured Petri nets [Jen97].

From a logical viewpoint, the verification procedure sketched above is the counterpart of a new, alternative, *bottom-up* semantics for a fragment of linear logic. The advantages of using a *backward* reachability procedure, as opposed to a *forward* one, are clearly advocated in [FS01, AJ01b]. In particular, computability results rely on the possibility of finding a *finite* representation for *infinite* set of states which are *upward-closed*. This can be done by specifying the *minimality requirements* that a given state should satisfy (remember the previous example, in which the multiset $\{use, use\}$ represents *every* configuration with *at least* two processes in state *use*). The resulting specification methodology turns out to be well-suited to specify an interesting class of *safety* properties (e.g. *mutual exclusion* in concurrency theory, *confidentiality* in authentication) for *parameterized* and *open* systems. From a logical perspective, this backward verification procedure relies on a *symbolic* evaluation of first-order programs based on *unification*, and makes use of suitable *subsumption* checks in order to relieve the *state-explosion* problem. In addition, techniques based on *static analysis* of programs provide further criteria to *prune* the search space.

Our framework has been intentionally designed as a *modular* (as opposed to *monolithic*) system. In fact, we have decided to leave the logical substratum, which basically consists of a multiset rewriting formalism, reasonably simple. The core of the system can then be enriched in different ways. For instance, reasoning on specialized and heterogeneous domains can be carried out by interfacing the system with different *constraint systems*, as in traditional constraint programming [JM94]. This solution is clearly more flexible, because the knowledge and reasoning capabilities which are specific of every particular domain can be delegated to specialized constraint solvers. In this way, we have been able to specify and analyze protocols which are both *parametric* and *infinite-state* (e.g. using integer-valued data variables).

As a result of our analysis, we have also obtained some results concerning *decidability* of the provability relation for some fragments of first-order linear logic (with or without constraints). From the verification viewpoint, these results are the counterpart of verification problems which, consequently, can be shown to be decidable. We have used these results as a termination guarantee for our verification procedure based on backward reachability. As an example, we have been able to analyze and validate, to our knowledge for the first time, a *parameterized* version of a classical *mutual-exclusion* protocol called the *ticket* protocol. It is remarkable that this protocol is both *parametric* and also *infinite-state*, in that it is

specified using integer variables which can get arbitrary values (even for a formulation of the protocol with a *fixed* number of processes, e.g. with *two* processes). Another interesting application field concerns *authentication protocols*. We will show that our methodology can be used either to find *attacks* or to verify correctness of security protocols. Our verification procedure, based on backward reachability, can *automatically* find attacks following complex patterns, like *parallel* attacks, *without encoding any prior knowledge* about the kind of attacks to look for.

Finally, we must mention that, from a logical standpoint, our work follows the established tradition of *fixpoint*, *bottom-up* semantics for (constraint) logic programs [Llo87, JM94, GDL95]. In particular, we have characterized the analogous of the so-called C-semantics [FLMP93, BGLM94], originally defined for Horn clause logic, for a subset of first-order linear logic including universal quantification in goals. As a side result, we have also obtained a non ground semantics for Horn programs enriched with universal quantification in goals. We believe that our work could also have an impact on the field of linear logic programming. This line of research can be carried on by considering either more complex linear logic languages [And92, HM94, Mil96, DM01], or more complex observational semantics, like the S-semantics of [FLMP93, BGLM94].

1.3 Plan of the Thesis

Broadly speaking, the thesis is divided into two parts. The first one presents the logical background for the rest of the thesis, and discusses the relationship between linear logic and classical alternative formalisms for the specification of concurrent systems, like (coloured) Petri nets and multiset rewriting systems.

The second part includes the original contribution of this thesis. Its core is structured as a collection of four chapters dealing with a different fragment or extension of the language LO [AP91b]. Every chapter deals with the definition of the bottom-up semantics for the corresponding fragment of linear logic, contains all the relevant proofs, and terminates with some examples. The four fragments or extensions of LO which we will be analyzed are: propositional LO; an extension of LO with the constant $\mathbf{1}$ and the $\otimes$ connective in goals, called $\mathrm{LO}_{\mathbf{1}}$; a first-order formulation of LO with constraints, called $\mathrm{LO}(\mathcal{C})$; and finally a first-order formulation of LO admitting universal quantification over goals, called $\mathrm{LO}_{\forall}$.

As a general guideline, we have included formal proofs and results stating correctness and completeness of the bottom-up semantics for each of the four different fragments mentioned above, even though some overlap may arise. For instance, the bottom-up semantics for propositional LO can be seen as an instance of the bottom-up semantics for $\mathrm{LO}_{\forall}$ programs. However, the semantics for propositional programs can be greatly simplified, and it also enjoys computability results which do not extend, in general, to the first-order case. We

also believe that the relevant proofs, which are much simpler in the propositional case, can help the reader to better understand the extensions presented in the following chapters. We have therefore chosen to include them in the corresponding chapter.

A more detailed description of the structure of the thesis is given below.

1.3.1 Structure of the Thesis

We describe below how the thesis is concretely structured. Some chapters are based or extend previously published works. For the convenience of the reader, a list of the relevant papers is given at the end of this section.

First of all, in **Chapter 2** we give some preliminaries and introduce some notations needed in the rest of the thesis.

The purpose of **Part I** is to present some background on linear logic and to substantiate our view of linear logic as a *unifying view of concurrency*. Specifically, this part is structured as follows. First of all, we introduce the basic concepts pertaining to linear logic in **Chapter 3**. In particular, we present in more detail the language LO and some proper extensions of its. In **Chapter 4** we discuss the relationship between linear logic, and in particular the language LO, and other formalisms for the specification of concurrent systems, namely Petri nets, multiset rewriting systems over atomic formulas, and broadcast protocols. In **Chapter 5** we extend the previous ideas by considering *first-order* linear logic theories and relating them to extensions of Petri nets, like coloured Petri nets, and first-order multiset rewriting systems.

Part II contains the original contribution of this thesis. In **Chapter 6**, first of all we discuss the *backward* verification approach which we follow in this thesis, we explain the connection with *bottom-up* semantics of linear logic programs, and we present a general theory which can be used to prove termination (decidability) results. **Chapter 7** deals with the propositional fragment of LO. The chapter is based on [BDM00, BDM01b, BDM02], and contains some simple examples illustrating our methodology. **Chapter 8** considers the language $\mathrm{LO}_{\mathbf{1}}$, an (apparently) simple extension of propositional LO which, however, has the effect of breaking the decidability property of the provability relation which holds instead for propositional LO. At the end of the chapter we show how broadcast primitives can be simulated in this logical fragment. The chapter extends previous results in [BDM00, BDM02]. In **Chapter 9** we consider the language $\mathrm{LO}(\mathcal{C})$, i.e., LO enriched with constraints. The analysis of the so-called *ticket* protocol, which we mentioned in Section 1.2, is included in this chapter. Some preliminary results concerning the contents of this chapter appeared in [BDM01a, BD02]. Finally, in **Chapter 10**, we extend the previous results to the fragment $\mathrm{LO}_{\forall}$, a first-order formulation of LO with universal quantification over goals. We present as example the analysis of a *test-and-lock* mutual exclusion proto-

col. **Chapter 11** presents some applications in the field of *security*, in particular it deals with the specification and analysis of *authentication* protocols, and should be considered as the logical prosecution of Chapter 10. Some preliminary results concerning the contents of this chapter have been presented in [Boz01].

Finally, we draw some conclusions in **Chapter 12**.

Publication List

[**BD02**] M. Bozzano and G. Delzanno. Beyond Parameterized Verification. In J.-P. Katoen and P. Stevens, editors, *Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 221–235, Grenoble, France, 2002. Springer-Verlag.

[**BDM00**] M. Bozzano, G. Delzanno, and M. Martelli. A Bottom-up Semantics for Linear Logic Programs. In *Proceedings 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 92–102, Montreal, Canada, 2000. ACM Press.

[**BDM01a**] M. Bozzano, G. Delzanno, and M. Martelli. An Effective Bottom-Up Semantics for First Order Linear Logic Programs. In H. Kuchen and K. Ueda, editors, *Proceedings 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 138–152. Springer-Verlag, Tokyo, Japan, 2001.

[**BDM01b**] M. Bozzano, G. Delzanno, and M. Martelli. On the Relations between Disjunctive and Linear Logic Programming. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming - Selected Papers from AGP 2000*, volume 48 of *ENTCS*, La Habana, Cuba, 2001.

[**BDM02**] M. Bozzano, G. Delzanno, and M. Martelli. An Effective Fixpoint Semantics for Linear Logic Programs. *Theory and Practice of Logic Programming*, 2(1):85–122, January 2002.

[**Boz01**] M. Bozzano. Ensuring Security through Model Checking in a Logical Environment (Preliminary Results). In *Proceedings of Workshop on Specification, Analysis and Validation for Emerging Technologies in Computational Logic (SAVE'01)*, Paphos,Cyprus, December 2001.

Chapter 2

Preliminaries

In this chapter we introduce some notions and terminology which will be used throughout the thesis.

2.1 Functions and Orders

We will use the notation $f : A \to B$ to denote a (total) **function** from the set A to the set B. $\mathbb{N}$ will denote the set of natural numbers, whereas $\mathbb{N}_0$ will stand for $\mathbb{N}\backslash\{0\}$, and $\mathbb{Z}$ will denote the set of integer numbers. A binary relation $\leqslant$ on a set A is said to be a **quasi-order** (or **preorder**) if it is reflexive and transitive. The pair $(A, \leqslant)$ is called a **preset**. If in addition $\leqslant$ is symmetric, it is an equivalence relation, while if it is antisymmetric it is a **partial order** and $(A, \leqslant)$ is called a **poset**. Given a poset $(A, \leqslant)$, an element $a \in A$ is an upper [lower] bound of a subset $X \subseteq A$ iff $x \leqslant a$ $[a \leqslant x]$ for all $x \in X$. The **least upper bound** (**l.u.b.**) of X, if it exists, is an upper bound of X which is minimum w.r.t. to $\leqslant$ among the set of upper bounds of X. Similarly, the **greatest lower bound** (**g.l.b.**) is defined as the maximum among the lower bounds of X. A **chain** is a function $C : \mathbb{N} \to A$ such that $C(i) \leqslant C(i+1)$ for all $i \in \mathbb{N}$. A poset A is called a **complete lattice** iff least upper bounds and greatest lower bounds exist for every (non empty) subset of A.

Given a complete lattice A with partial ordering $\leqslant$, l.u.b. $\bigsqcup$, and least element $\bot$, a function $f : A \to A$ is said to be **monotonic** iff $a \leqslant b$ implies $f(a) \leqslant f(b)$ for all $a, b \in A$; f is **continuous** iff for every chain $C : \mathbb{N} \to A$ we have that $f(\bigsqcup C) = \bigsqcup(f(C))$, where it is meant that $\bigsqcup C$ stands for $\bigsqcup_{i\in\mathbb{N}} C(i)$ and $f(C)$ is the chain such that $f(C)(i) = f(C(i))$ for every $i \in \mathbb{N}$. A **fixpoint** for a function $f : A \to A$ is any element $x \in A$ such that $f(x) = x$. By Knaster-Tarski fixpoint theorem [Llo87], the set of fixpoints of any monotonic function $f : A \to A$ on a complete lattice A is a non empty complete lattice, and therefore f admits

a **least fixpoint**, denoted $lfp(f)$, and a **greatest fixpoint**, denoted $gfp(f)$. If in addition f is continuous, it holds that $lfp(f) = f{\uparrow}_\omega$, where for $k \in \mathbb{N}$, $f{\uparrow}_k$ and $f{\uparrow}_\omega$ are defined as follows: $f{\uparrow}_0 = \bot$, $f{\uparrow}_{k+1} = f(f{\uparrow}_k)$ for all $k \geq 0$, and $f{\uparrow}_\omega = \bigsqcup_{k\in\mathbb{N}} f{\uparrow}_k$.

2.2 Multisets

Throughout the thesis, we will extensively use the concept of **multiset** (or **bag**). Intuitively, a multiset extends the notion of set by allowing multiple occurrences of the same element. Formally, a multiset with elements in D is a function $\mathcal{M} : D \to \mathbb{N}$. If $d \in D$ and $\mathcal{M}$ is a multiset on D, we say that $d \in \mathcal{M}$ if and only if $\mathcal{M}(d) > 0$. For convenience, we will often use the notation for sets (allowing duplicated elements) to indicate multisets, when no ambiguity arises from the context. For instance, $\{a, a, b\}$, where $a, b \in D$, will denote the multiset $\mathcal{M}$ such that $\mathcal{M}(a) = 2$, $\mathcal{M}(b) = 1$, and $\mathcal{M}(d) = 0$ for all $d \in D\backslash\{a, b\}$. Sometimes we will simply write a, a, b for $\{a, a, b\}$. Finally, given a set D, $\mathcal{MS}(D)$ will denote the set of multisets with elements in D. We define the following operations on multisets.

Definition 2.1 (Operations on Multisets) *Let D be a set, $\mathcal{M}_1, \mathcal{M}_2 \in \mathcal{MS}(D)$, and $n \in \mathbb{N}$.*

i. ϵ *is s.t.* $\epsilon(d) = 0$ *for all* $d \in D$ (**empty multiset**)*;*

ii. $(\mathcal{M}_1 + \mathcal{M}_2)(d) = \mathcal{M}_1(d) + \mathcal{M}_2(d)$ *for all* $d \in D$ (**union**)*;*

iii. $(\mathcal{M}_1 \backslash \mathcal{M}_2)(d) = max\{0, \mathcal{M}_1(d) - \mathcal{M}_2(d)\}$ *for all* $d \in D$ (**difference**)*;*

iv. $(\mathcal{M}_1 \cap \mathcal{M}_2)(d) = min\{\mathcal{M}_1(d), \mathcal{M}_2(d)\}$ *for all* $d \in D$ (**intersection**)*;*

v. $(n \cdot \mathcal{M})(d) = n\mathcal{M}(d)$ *for all* $d \in D$ (**scalar product**)*;*

vi. $\mathcal{M}_1 \neq \mathcal{M}_2$ *iff there exists* $d \in D$ *s.t.* $\mathcal{M}_1(d) \neq \mathcal{M}_2(d)$ (**comparison**)*;*

vii. $\mathcal{M}_1 \preccurlyeq \mathcal{M}_2$ *iff* $\mathcal{M}_1(d) \leq \mathcal{M}_2(d)$ *for all* $d \in D$ (**inclusion**)*;*

viii. $(\mathcal{M}_1 \bullet \mathcal{M}_2)(d) = max\{\mathcal{M}_1(d), \mathcal{M}_2(d)\}$ *for all* $d \in D$ (**merge**)*;*

ix. $|\mathcal{M}_1| = \Sigma_{d\in D}\mathcal{M}_1(d)$ (**cardinality**).

We will use the summation notation $\sum_{i \in I} \mathcal{M}_i$ to denote the union of a family of multisets $\mathcal{M}_i$, with $i \in I$, I being a finite set. It turns out that $(\mathcal{MS}(D), \preccurlyeq)$ has the structure of a lattice (the lattice is complete provided a greatest element is added). In particular, merge and intersection are, respectively, the least upper bound and the greatest lower bound operators with respect to multiset inclusion $\preccurlyeq$.

2.3 Signatures and Algebras

In the following we will often need **signatures** to keep trace of constant, function, and possibly predicate symbols (in this case we will say "signature with predicates") used in programs. We always assume the sets of constant, function, and predicate symbols to be mutually disjoint.

Given a signature Σ, we take for granted the definitions of Σ-**algebra** and **term algebra** over Σ. A term is said to be **ground** if it does not contain variables. Given a denumerable set of variable symbols $\mathcal{V}$ (usually noted x, y, z, ...), we denote by $T_\Sigma^\mathcal{V}$ the set of *non ground* terms over Σ and $\mathcal{V}$, while T_Σ denotes the set of *ground* terms over Σ. We use the notation $t[s/x]$ to indicate the *capture-free* substitution of the term s for the variable $x \in \mathcal{V}$ in a term t. We extend this notation to $t[\mathbf{s}/\mathbf{x}]$, where $\mathbf{s}$ and $\mathbf{x}$ are, respectively, a vector of terms and a vector of variables, with the same number of components. Finally, we denote by $A_\Sigma^\mathcal{V}$ (A_Σ) the set of non ground (ground) atoms over Σ.

In Chapter 10 we will need to augment signatures with newly created constants, called *eigenvariables*. We will use the compact notation Σ, c to denote the augmentation of Σ by c.

2.4 Substitutions and Multiset Unifiers

We inherit the usual concept of **substitution** (mapping from variables to terms) from traditional logic programming. We will always consider a denumerable set of variables $\mathcal{V}$, and substitutions will be usually noted θ, σ, τ, ... We will use the notation $[x \mapsto t, \ldots]$, where x is a variable and t is a term, to denote substitution bindings, with *nil* denoting the empty substitution. The application of a substitution θ to F, where F is a generic expression (e.g. a formula, a term, ...) will be denoted by $F\theta$. A substitution θ is said to be **grounding** for F if $F\theta$ is ground, in this case $F\theta$ is called a **ground instance** of F. Composition of two substitutions θ and σ will be denoted $\theta \circ \sigma$, e.g. $F(\theta \circ \sigma)$ stands for $(F\theta)\sigma$. We will indicate the domain of a substitution θ by $Dom(\theta)$, and will say "θ defined on a signature Σ" meaning that θ can only map variables in $Dom(\theta)$ to terms in $T_\Sigma^\mathcal{V}$. Substitutions are ordered with respect to the ordering $\leq$ defined in this way: $\theta \leq \tau$ if and only if there exists a substitution σ s.t. $\tau = \theta \circ \sigma$. If $\theta \leq \tau$, θ is said to be **more general** than τ; if $\theta \leq \tau$ and $\tau \leq \theta$, θ and τ are said to be **equivalent**. Finally, $FV(F)$, for an expression F, will denote the set of *free* variables of F, and $\theta_{|W}$, where $W \subseteq \mathcal{V}$, will denote the *restriction* of θ to $Dom(\theta) \cap W$.

We will need the notion of **most general unifier** (*m.g.u.*). The definition of most general unifier is somewhat delicate. In particular, different classes of substitutions (e.g. idempo-

tent substitutions) have been considered for defining most general unifiers. We refer the reader to [Ede85, LMM88, Pal90] for a discussion. Most general unifiers form a complete lattice with respect to the ordering $\leq$, provided a greatest element is added. For our purposes, we do not choose a particular class of most general unifiers, we only require the operation of **least upper bound** of two substitutions w.r.t $\leq$ to be defined and effective. The least upper bound of θ_1 and θ_2 (see [Pal90] for its definition) will be used in Chapter 10 and indicated $\theta_1 \uparrow \theta_2$. We assume $\uparrow$ to be commutative and associative.

Example 2.2 Let us consider a signature with a constant symbol a and a function symbol f. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$. Let $\theta_1 = [x \mapsto u, y \mapsto f(u')]$, $\theta_2 = [x \mapsto v, y \mapsto f(a)]$, and $\theta_3 = [x \mapsto f(w), y \mapsto z]$. Then we have that $\theta_1 \leq \theta_2$, θ_1,θ_3 and θ_2,θ_3 are not comparable w.r.t. $\leq$, and, modulo renaming, $\theta_1 \uparrow \theta_2 \uparrow \theta_3 = [x \mapsto f(x'), y \mapsto f(a)]$. □

Multiset Unifiers. We need to lift the definition of *most general unifier* from expressions to multisets of expressions. Namely, given two multisets $\mathcal{A} = \{a_1, \ldots, a_n\}$ and $\mathcal{B} = \{b_1, \ldots, b_n\}$ (note that $|\mathcal{A}| = |\mathcal{B}|$), we define a most general unifier of $\mathcal{A}$ and $\mathcal{B}$, written $m.g.u.(\mathcal{A}, \mathcal{B})$, to be the most general unifier (defined in the usual way) of the two vectors of expressions $\langle a_1, \ldots, a_n \rangle$ and $\langle b_{i_1}, \ldots, b_{i_n} \rangle$, where $\{i_1, \ldots, i_n\}$ is a permutation of $\{1, \ldots, n\}$. Depending on the choice of the permutation, in general there will be more than one way to unify two given multisets (the resulting class of m.g.u. in general will include unifiers which are not equivalent). We will use the notation $\theta = m.g.u.(\mathcal{A}, \mathcal{B})$ to denote any unifier which is *non deterministically* picked from the set of most general unifiers of $\mathcal{A}$ and $\mathcal{B}$.

Example 2.3 Let us consider a signature with a constant symbol a, a function symbol f and a predicate symbol p. Let $\mathcal{V}$ be a a denumerable set of variables, and $v, w, x, \ldots \in \mathcal{V}$. Let $\mathcal{A} = \{p(x, x), p(f(y), y)\}$ and $\mathcal{B} = \{p(v, a), p(w, z)\}$. The multisets $\mathcal{A}$ and $\mathcal{B}$ can be unified in two different ways. By unifying $p(x, x)$ with $p(v, a)$ and $p(f(y), y)$ with $p(w, z)$, we get, modulo renaming, the m.g.u. $\theta_1 = [x \mapsto a, v \mapsto a, w \mapsto f(y), z \mapsto y]$. By switching the order of atom unifications, we get instead, again modulo renaming, the m.g.u. $\theta_2 = [w \mapsto x, z \mapsto x, v \mapsto f(a), y \mapsto a]$. As it can be readily verified, neither $\theta_1 \leq \theta_2$ nor $\theta_2 \leq \theta_1$ hold. We also have that $\mathcal{A}\theta_1 = \mathcal{B}\theta_1 = \{p(a, a), p(f(y), y)\}$, while $\mathcal{A}\theta_2 = \mathcal{B}\theta_2 = \{p(x, x), p(f(a), a)\}$. □

2.5 Basics of Logic Programming

We recall here some basic notions in Logic Programming (see for instance [Llo87, Apt90]).

Syntax. A (positive or negative) **literal** is an atom or the negation of an atom. A **clause** is a disjunction of literals, the set of positive literals is called the **head** and the set of negative literals the **body**. Clauses with at most one positive literal are called **Horn clauses**, a clause with exactly one positive literal is called **program clause** or **definite clause**, and usually noted $H \leftarrow B_1, \ldots, B_n$, where H is the head and $B_1, \ldots, B_n$ the body literals (stripped of the negation symbol). A definite clause with empty body is called **unit clause**, whereas a Horn clause with empty head is called **goal** or **negative clause**. A **logic program** is a finite and non empty set of program clauses.

Semantics. Given a program P over a signature Σ, with non empty set of constants, the **Herbrand universe** of P is the set of all ground terms over Σ, while the **Herbrand base** of P is the set of ground atoms over Σ. By **Herbrand interpretation** we mean an interpretation for P such that its domain is the Herbrand universe, constants are assigned to themselves, and every n-ary function symbol f is assigned to the function mapping any sequence of terms $t_1, \ldots, t_n$ to $f(t_1, \ldots, t_n)$. A Herbrand interpretation is uniquely determined by a subset of the Herbrand base fixing the interpretation of predicate symbols.

A logic program P can be equipped with three kinds of semantics:

- **operational semantics**, usually given via the notion of SLD-resolution (see [Llo87, Apt90]) or via a suitable *proof-system*;
- **model-theoretic semantics**, defined via the notion of *satisfiability* of a program w.r.t. a given (Herbrand) *interpretation*;
- **fixpoint semantics**, given by the least fixpoint of a suitable **immediate consequence operator**, usually denoted T_P.

A suitable notion of *equivalence* between the three different kinds of semantics can be proved [Llo87, Apt90]. In the rest of the thesis, we will be mainly interested in the fixpoint semantics. The T_P operator maps Herbrand interpretations (i.e., subsets of the Herbrand base) to Herbrand interpretations, and is defined as follows: $H \in T_P(I)$ if and only if $H \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P, and $B_1, \ldots, B_n \in I$. T_P can proved to be monotonic and continuous, therefore it admits a least fixpoint $lfp(T_P) = T_P{\uparrow}_\omega$, which coincides with the **least Herbrand model** of P.

We conclude by mentioning that the above scheme can be extended in order to take into consideration more refined notions of *observables* [FLMP93, BGLM94]. In particular, the so-called *C-semantics* formalizes the notion of *non-ground success set* of a logic program, whereas the so-called *S-semantics* correspond to the *computed answer substitution semantics*. The C-semantics is defined by considering the *non-ground* Herbrand base (i.e., the set of non-ground atoms over a signature Σ) and the corresponding notion of *non-ground*

Herbrand interpretation. The operational semantics is defined as the set of non-ground atoms which have an *SLD-refutation* with *empty answer substitution* (see [Llo87, Apt90]). An extended T_P operator can be defined, which, as usual, turns out to be monotonic and continuous, therefore admitting a least fixpoint $lfp(T_P) = T_P{\uparrow}_\omega$.

In Chapter 10 we will extend the C-semantics of [FLMP93, BGLM94] to a subset of linear logic programs.

Part I

Linear Logic as a Unifying View of Concurrency

Chapter 3

Linear Logic

Linear logic [Gir87] has been introduced as a resource-oriented refinement of classical logic. Intuitively, the idea is to constrain the number of times a given assumption (resource *occurrence*) can be used inside a deduction for a given goal formula. This peculiar resource management together with the possibility of naturally modeling the notion of *state*, make linear logic an appealing formalism to reason about *concurrent* and *dynamically changing* systems.

After giving a very brief summary of some aspects related to linear logic and linear logic programming in Section 3.1, in Section 3.2 we will focus on the language LO [AP91b] and some of its extensions, which will be the leading thread of the present thesis work.

3.1 Full Linear Logic

In classical logic one usually thinks about the construction of a proof as the process of finding a logical deduction for a goal starting from a given *set* of assumptions. In linear logic one may think of assumptions as *multisets* of *resources*, each of which can be used just once. This view of assumptions imposes a shift in the concept of proof as well. Building a proof in linear logic is the process of finding a deduction which *transforms* all the given assumptions (and only them) into something else (the conclusion). This view of proof construction is related to the notion of *causal implication*. To exemplify, a reaction in chemistry is a typical example of a process that can be naturally modeled in linear logic (everybody knows that reactions must be correctly *balanced*).

Another feature which is intrinsic to linear logic is the possibility of modeling concurrent executions (e.g. chemical reactions which happen simultaneously). This is witnessed by several papers relating linear logic with different formalisms for concurrency, e.g. [EW90,

classical	t	f	$\wedge$	$\vee$	$\supset$
multiplicative	$\mathbf{1}$	$\bot$	$\otimes$	$\parr$	$\multimap$
	one	anti	tensor	par	lolli
additive	$\top$	0	$\&$	$\oplus$	$\rightsquigarrow$
	all	zero	with	plus	wave
exponentials		$!$		$?$	
		bang		why not	

Table 3.1: The set of linear logic connectives

ALPT93, Mil92, Cer94, Laf95, CDKS00]. For instance, in [Mil92] a translation of π-calculus into linear logic is presented. The translation yields a proof system in linear logic whose provability relation is the counterpart of the π-calculus reduction relation, and can be used, e.g., to prove *structural equivalence* of π-calculus agent expressions.

Linear logic can be considered as the endpoint of a proof-theoretical investigation of subsystems of classical logic. In particular, the analysis involves the so called *structural rules*, i.e., *exchange*, *weakening* and *contraction*. In fact, the *weakening* and *contraction* rules are not compatible with the notion of resource sketched above: the weakening rule allows one to ignore a given assumption, while the contraction rule is responsible for duplication of assumptions. Removing the structural rules of weakening and contraction from classical sequent calculus has the unavoidable consequence of splitting the set of classical connectives into two different classes, called *additive* and *multiplicative*. In the usual sequent calculus presentation, additive and multiplicative connectives differ with respect to context management (context *sharing* vs. context *splitting*, respectively). While in classical logic both presentations, additive and multiplicative, are provably equivalent, the removal of structural rules makes them not equivalent anymore.

Although removing the structural rules seems to be unavoidable in this setting, it would be unreasonable to remove them altogether. In other words, a way to recover the power of classical logic is necessary. To this aim, two further connectives, the so-called *exponentials*, one dual with the other, are introduced. Intuitively, every formula preceded by an exponential behaves like the corresponding classical one (e.g. an hypothesis can be used any number of times or not used at all in a proof). This mechanism re-introduces classical connectives in a controlled way. A summary of linear logic connectives is presented in Table 3.1. We note that it is also possible to define a linear negation (denoted $\cdot^\perp$ in the following), which turns out to be *involutive* (i.e., $(F^\perp)^\perp = F$) and constructive at the same time.

In Figure 3.1 we present an example of a *sequent* system for full linear logic. It is an

$$\frac{}{\vdash \Theta : F, F^{\perp}}\ id \qquad \frac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Delta, F^{\perp}}{\vdash \Theta : \Gamma, \Delta}\ cut \qquad \frac{\vdash \Theta, F : \Gamma, F}{\vdash \Theta, F : \Gamma}\ abs$$

$$\frac{\vdash \Theta : \Gamma}{\vdash \Theta : \Gamma, \bot}\ \bot \qquad \frac{\vdash \Theta : \Gamma, F, G}{\vdash \Theta : \Gamma, F \parr G}\ \parr \qquad \frac{\vdash \Theta, F : \Gamma}{\vdash \Theta : \Gamma, ?F}\ ?$$

$$\frac{}{\vdash \Theta : \mathbf{1}}\ \mathbf{1} \qquad \frac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Delta, G}{\vdash \Theta : \Gamma, \Delta, F \otimes G}\ \otimes \qquad \frac{\vdash \Theta : F}{\vdash \Theta : !F}\ !$$

$$\frac{}{\vdash \Theta : \Gamma, \top}\ \top \qquad \frac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Gamma, G}{\vdash \Theta : \Gamma, F \,\&\, G}\ \& \qquad \frac{\vdash \Theta : \Gamma, F[c/x]}{\vdash \Theta : \Gamma, \forall x.F}\ \forall$$

$$\frac{\vdash \Theta : \Gamma, F}{\vdash \Theta : \Gamma, F \oplus G}\ \oplus_l \qquad \frac{\vdash \Theta : \Gamma, G}{\vdash \Theta : \Gamma, F \oplus G}\ \oplus_r \qquad \frac{\vdash \Theta : \Gamma, F[t/x]}{\vdash \Theta : \Gamma, \exists x.F}\ \exists$$

Figure 3.1: A one-sided, dyadic proof system for full linear logic

example of *one-sided* sequent system (*two-sided* versions are possible [Tro92]). It is also a *dyadic* sequent system, according to the terminology of [And92]. In fact, sequents of the form $\vdash \Theta : \Gamma$ are divided into two parts Θ and Γ, which are *multisets* of formulas. Θ is the so-called *unbounded* part, while Γ is the *bounded* part. In other words, formulas in Θ must be *implicitly* considered as *exponentiated* (i.e., preceded by ?) and thus can be reused any number of times, while formulas in Γ must be used exactly once. The different behaviour of multiplicative rules w.r.t. additive rules (i.e., context splitting vs. context sharing) should be evident. For instance, the reader can compare the different context management in rules $\otimes$ and $\&$. In rule $\forall$, c stands for a new constant, while in rule $\exists$, t stands for an arbitrary term. The rules for implications are missing but can be recovered by means of logical equivalences like $F \multimap G \equiv F^{\perp} \parr G$.

3.1.1 Programming in Linear Logic

In recent years a number of fragments of linear logic have been proposed as a logical foundation for extensions of logic programming. Several new programming or specification languages, like LO [AP91b] (see Section 3.2), LinLog [And92], ACL [KY95], Lolli [HM94], Lygon [HP94], Forum [Mil96] and $\mathcal{E}_{hhf}$[DM01] (a subset of Forum) have been proposed with the aim of enriching traditional logic programming languages with a well-founded notion of state and with aspects of concurrency. For instance, LO has been applied to model coordination languages [And96], Forum has been used for specifying the sequential and

pipelined operational semantics of the DLX machine [Chi95], and $\mathcal{E}_{hhf}$ to model object-oriented and deductive databases [BDM97], multi-agent systems [BDM$^+$99], and object calculi [BDLM00].

Further research efforts have tried to incorporate linear logic within logical frameworks. A *logical framework* can be seen as a meta-language for specifying and reasoning about different formalisms represented as deductive systems, such as various logics, natural deduction inference systems and sequent calculi, programming languages, abstract machines, and so on. In [MMP96, McD97], the authors present a logical framework based on extensions of linear logic and intuitionistic logic incorporating a notion of *definition* and induction on natural numbers. Such framework has been applied, for instance, to prove simulation and bisimulation judgments in abstract transition systems, and properties such as subject reduction and determinacy of evaluation in the language PCF. A different approach is the one followed in [Pfe01], where a logical framework is presented, based on a constructive type theory incorporating *dependent types*, according to the so-called *judgments-as-types* representation. These ideas have been incorporated in the system Twelf [PS99] and exploited to prove, for instance, various properties in the area of Mini-ML, natural deduction and category theory. A linear logical extension is considered in [CP02].

The operational semantics of linear logic languages is usually given via a sequent-calculi presentation of the corresponding fragment of linear logic. Special classes of proofs have been studied in order to limit the non-determinism intrinsic in the process of proof construction. In particular, the *focusing* proofs of [And92] and the *uniform* proofs of [Mil96] allow one to consider *cut-free* and *goal-driven* proof systems that are still complete with respect to provability in linear logic.

The key issue underlying the notion of goal-driven provability is the concept of *synchronous* and *asynchronous* connectives [And92]. Asynchronous connectives ($\bot$, $\parr$, ?, $\top$, &, $\forall$) require no choice from the proof search procedure, and only introduce *don't care* non-determinism, while synchronous connectives ($\mathbf{1}$, $\otimes$, !, 0, $\oplus$, $\exists$) require the search procedure to make a committed choice and introduce *don't know* non-determinism. Specifically, the focusing proofs of [And92] have the following features:

- if the current goal contains some asynchronous formulas, they are immediately decomposed: any of them can be *immediately* and *randomly* selected as the principal formula for decomposition;
- if the current goal contains only synchronous formulas, one of them must be selected *non deterministically* for processing, but as soon as this choice is made, the proof can *focus* on it, i.e., it can strip all layers of synchronous connectives from it.

The above restrictions on the proof search procedure induce a proof normalization which does not affect completeness of the provability relation. In order to enforce this behaviour

of the proof search procedure, a variant of the proof system of Figure 3.1 based on focusing proofs (called *triadic*) is presented in [And92].

When dealing with specific languages based on linear logic, the focusing proofs of [And92] can be furtherly refined and tailored to the particular logical fragment under consideration. For instance, focusing proofs have a simple and direct computational interpretation for the fragment called LinLog, presented in [And92]. LinLog is based on a subset of linear logic for which a sound mapping from full linear logic, preserving focusing proofs, exists. LinLog formulas extend the syntax of definite clauses and goals of traditional Horn logic, by allowing clauses with multiple atoms (connected by ⅋) in the head, and goals built over a subset of linear connectives. The computational interpretation of LinLog is particularly suited for concurrency and object-oriented programming (program clauses are called *methods* in [And92]).

In this thesis, we will focus our attention on a fragment of LinLog called LO [AP91b]. The reason we selected LO is that we were looking for a relatively simple linear logic language with a focusing-proof presentation, state-based computations and aspects of concurrency. As we will demonstrate in Chapter 4, LO programs are at least as expressive as Petri nets or multiset rewriting systems over atomic formulas. In practice, LO has been successfully applied to model, e.g., concurrent object-oriented languages [AP91b], and coordination languages based on the Linda model [And96]. In this thesis we will show that LO, enriched with a computational model based on *bottom-up* evaluation, can provide a valuable environment for the specification and validation of protocols.

For more details on the material of this section, we refer the reader to [Gir87, Tro92, CD97] for an introduction to linear logic, [MNPS91] and [Mil96] for a discussion on, respectively, *uniform* proofs in intuitionistic and in linear logic, and [And92] for the related concept of *focusing* proofs and for LinLog. Some background material on classical logic and sequent calculus can be found in [Gal86]. In the next section we present the language LO.

3.2 The Language LO

LO [AP91b] is a logic programming language based on a fragment of LinLog [And92]. Its mathematical foundations lie on a proof-theoretical presentation of a fragment of linear logic defined over the linear connectives $\multimap$ (*linear implication*, we use the reversed notation $H \circ\!\!- G$ for $G \multimap H$), & (*additive conjunction*), ⅋ (*multiplicative disjunction*), and the constant $\top$ (*additive identity*). In this section we present the proof-theoretical semantics of LO, which corresponds to the usual *top-down* operational semantics for traditional logic programming languages like Prolog. For the sake of precision, we present a slight extension of LO by admitting the constant $\bot$ in goals and clause heads. Following [AP91b], we give these definitions.

Definition 3.1 (Atomic Formulas) *Let Σ be a signature with predicates including a set of constant and function symbols $\mathcal{L}$ and a set of predicate symbols $\mathcal{P}$, and let $\mathcal{V}$ be a denumerable set of variables. An atomic formula over Σ and $\mathcal{V}$ has the form $p(t_1, \ldots, t_n)$ (with $n \geq 0$), where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are (non ground) terms in $T_\Sigma^\mathcal{V}$.*

We are now ready to define **G**-formulas, which correspond to *goals* to be evaluated in a given program.

Definition 3.2 (Goal Formulas) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. The class of* **G***-formulas (goal formulas) over Σ and $\mathcal{V}$ is defined by the following grammar*

$$\mathbf{G} \ ::= \ \mathbf{G} \parr \mathbf{G} \ | \ \mathbf{G} \ \& \ \mathbf{G} \ | \ \mathbf{A} \ | \ \top \ | \ \bot$$

where **A** *stands for an atomic formula over Σ and $\mathcal{V}$. A multiset of goal formulas will be called a* **context** *hereafter.*

The following definitions introduce **D**-formulas, which correspond to multiple-headed program clauses.

Definition 3.3 (Head Formulas) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. The class of* **H***-formulas (head formulas) over Σ and $\mathcal{V}$ is defined by the following grammar*

$$\mathbf{H} \ ::= \ \mathbf{A} \parr \ldots \parr \mathbf{A} \ | \ \bot$$

where **A** *stands for an atomic formula over Σ and $\mathcal{V}$.*

Definition 3.4 (Clauses and Programs) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. The class of* **D***-formulas (program clauses) over Σ and $\mathcal{V}$ is defined by the following grammar*

$$\mathbf{D} \ ::= \ \forall(\mathbf{H} \ \multimap \ \mathbf{G}) \ | \ \mathbf{D} \ \& \ \mathbf{D}$$

where **H** *and* **G** *are, respectively, a head formula and a goal formula over Σ and $\mathcal{V}$, and $\forall(H \multimap G)$ stands for $\forall x_1 \ldots x_k.(H \multimap G)$, with $\{x_1, \ldots, x_k\} = FV(H \multimap G)$. An LO program over Σ and $\mathcal{V}$ is a* **D***-formula over Σ and $\mathcal{V}$.*

For the sake of simplicity, in the following we usually omit universal quantifiers in program clauses, i.e., we consider free variables as being *implicitly* universally quantified.

Definition 3.5 (LO Sequents) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. An LO sequent has the form $P \vdash G_1, \ldots, G_k$, where $P = D_1 \,\&\, \ldots \,\&\, D_n$ is an LO program and $G_1, \ldots, G_k$ is a context, i.e., a multiset of goals, over Σ and $\mathcal{V}$.*

Remark 3.6 Given an LO program $P = D_1 \,\&\, \ldots \,\&\, D_n$, in the rest of the thesis we often find it convenient to view P as the *set* of clauses $D_1, \ldots, D_n$. Formally, this is justified by the logical equivalence between $!(D_1 \,\&\, \ldots \,\&\, D_n)$ and $!D_1 \otimes \ldots \otimes !D_n$.

According to remark 3.6, the left-hand side of an LO sequent can be seen as the *set* of clauses $D_1, \ldots D_n$, while the right-hand side is a *multiset* of goals. Structural rules (*exchange*, *weakening* and *contraction*) are allowed on the left-hand side, while on the right-hand side only the rule of exchange is allowed (for the fragment under consideration, it turns out that the rule of weakening is admissible, as stated in Proposition 3.9, while contraction is forbidden). The LO sequent $P \vdash G_1, \ldots, G_k$ stands for the following one-sided linear logic sequent (see Figure 3.1):

$$\vdash {D_1}^{\perp}, \ldots, {D_n}^{\perp} : G_1, \ldots, G_k.$$

We remind that the formulas on the left of ':' can be used in a proof an arbitrary number of times. In other words, an LO program can also be viewed as a set of *reusable* clauses.

We now define provability in LO.

Definition 3.7 (Ground Instances) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. Given an LO program P over Σ and $\mathcal{V}$, the set of ground instances of P, denoted $Gnd(P)$, is defined as follows:*

$$Gnd(P) = \{(H \multimap G)\,\theta \mid \forall\, (H \multimap G) \in P \text{ and } \theta \text{ is a grounding substitution for } H \multimap G\}$$

The execution of a multiset of **G**-formulas $G_1, \ldots, G_k$ in P corresponds to a *goal-driven* proof for the two-sided LO sequent $P \vdash G_1, \ldots, G_k$. According to this view, the operational semantics of LO is given via the *uniform* (*focusing*) proof system presented in Figure 3.2, where P is a set of clauses, $\mathcal{A}$ is a multiset of atomic formulas, and Δ is a multiset of **G**-formulas. We have used the notation $\widehat{H}$, where H is a linear disjunction of atomic formulas $a_1 \parr \ldots \parr a_n$, to denote the multiset $a_1, \ldots, a_n$ (by convention, $\widehat{\perp} = \epsilon$, where ϵ is the empty multiset).

Definition 3.8 (LO provability) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. Given an LO program P and a goal G, over Σ and $\mathcal{V}$, we say that G is provable from P if there exists a proof tree, built over the proof system of Figure 3.2, with root $P \vdash G$, and such that every branch is terminated with an instance of the $\top_r$ axiom.*

The concept of *uniformity* applied to LO requires that the right rules $\top_r, \parr_r, \&_r, \perp_r$ have priority over *bc*, i.e., *bc* is applied only when the right-hand side of a sequent is a multiset of *atomic* formulas (as suggested by the notation $\mathcal{A}$ in Figure 3.2). The proof system of

$$\frac{}{P \vdash \top, \Delta}\ \top_r \qquad \frac{P \vdash G_1, G_2, \Delta}{P \vdash G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash G_1, \Delta \quad P \vdash G_2, \Delta}{P \vdash G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash \Delta}{P \vdash \bot, \Delta}\ \bot_r \qquad \frac{P \vdash G, \mathcal{A}}{P \vdash \widehat{H}, \mathcal{A}}\ bc\ \ (H \circ\!\!- G\ \in Gnd(P))$$

Figure 3.2: A proof system for LO

Figure 3.2 is a specialization of more general uniform proof systems for linear logic like Andreoli's focusing proofs [And92] and Forum [Mil96]. Rule *bc* is analogous to a backchaining (resolution) step in traditional logic programming languages. Note that according to the concept of resolution explained above, *bc* can be executed only if the right-hand side of the current LO sequent consists of atomic formulas. As an instance of rule *bc*, we get the following proof fragment, which deals with the case of clauses with empty head:

$$\frac{\begin{array}{c}\vdots\\ P \vdash \mathcal{A}, G\end{array}}{P \vdash \mathcal{A}}\ bc$$

$$provided\ \bot \circ\!\!- G \in Gnd(P)$$

Given that clauses with empty head are always applicable in atomic *contexts*, the degree of non-determinism they introduce in proof search is usually considered unacceptable [Mil96] and in particular they are forbidden in the original presentation of LO [AP91b]. However, the computational model we are interested in, i.e., bottom-up evaluation, does not suffer this drawback. Clauses with empty head often allow more flexible specifications (see for instance Chapter 11).

LO clauses having the form $H \circ\!\!- \top$ play the same role as axioms (i.e., unit clauses) for Horn programs. In fact, when a backchaining step over such a clause is possible, we get a *successful* (branch of) computation, independently of the current *context* $\mathcal{A}$, as shown in the following proof scheme:

$$\frac{\dfrac{}{P \vdash \top, \mathcal{A}}\ \top_r}{P \vdash \widehat{H}, \mathcal{A}}\ bc$$

$$provided\ H \circ\!\!- \top \in Gnd(P)$$

This observation is formally stated in the following proposition (we recall that $\preccurlyeq$ is the multiset inclusion relation).

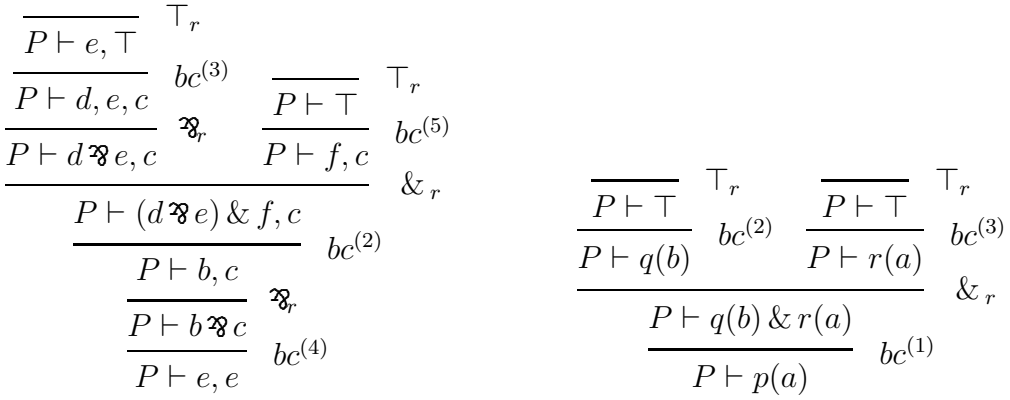

Figure 3.3: Examples of LO proofs

Proposition 3.9 (Admissibility of the Weakening Rule) *Given an LO program P and two multisets of goals Δ, Δ' such that $\Delta \preccurlyeq \Delta'$, if $P \vdash \Delta$ then $P \vdash \Delta'$.*

Proof By simple induction on the structure of LO proofs. □

Admissibility of the weakening rule makes LO an *affine* fragment of linear logic [Kop95]. Note that all structural rules are admissible on the left hand side (i.e., on the program part) of LO sequents.

Example 3.10 Let P be the (propositional) LO program consisting of the clauses

1. $a \circ\!\!- b \parr c$
2. $b \circ\!\!- (d \parr e) \,\&\, f$
3. $c \parr d \circ\!\!- \top$
4. $e \parr e \circ\!\!- b \parr c$
5. $c \parr f \circ\!\!- \top$

where a, b, c, d, e, f are propositional symbols, and consider an initial goal e, e. A proof for this goal is shown on the left-hand side of Figure 3.3, where we have denoted by $bc^{(i)}$ the application of the backchaining rule over clause number i of P. The proof proceeds as follows. Using clause 4, to prove e, e we have to prove $b \parr c$, which, by LO $\parr_r$ rule, reduces to prove b, c. At this point we can backchain over clause 2, and we get the new goal $(d \parr e) \,\&\, f, c$. By applying $\&_r$ rule, we get two separate goals $d \parr e, c$ and f, c. The first, after a reduction via $\parr_r$ rule, is provable by means of clause (axiom) 3, while the latter is provable directly by clause (axiom) 5. Note that $\top$ succeeds in a non-empty context

$$\frac{}{P \vdash_{\mathbf{1}} \top, \Delta}\ \top_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, G_2, \Delta}{P \vdash_{\mathbf{1}} G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, \Delta \quad P \vdash_{\mathbf{1}} G_2, \Delta}{P \vdash_{\mathbf{1}} G_1 \,\&\, G_2, \Delta}\ \&_r \qquad \frac{P \vdash_{\mathbf{1}} \Delta}{P \vdash_{\mathbf{1}} \bot, \Delta}\ \bot_r$$

$$\frac{}{P \vdash_{\mathbf{1}} \mathbf{1}}\ \mathbf{1}_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, \Delta_1 \quad P \vdash_{\mathbf{1}} G_2, \Delta_2}{P \vdash_{\mathbf{1}} G_1 \otimes G_2, \Delta_1, \Delta_2}\ \otimes_r \qquad \frac{P \vdash_{\mathbf{1}} G, \mathcal{A}}{P \vdash_{\mathbf{1}} \widehat{H}, \mathcal{A}}\ bc\ \ (H \multimap G\ \in Gnd(P))$$

Figure 3.4: A proof system for $\mathrm{LO}_{\mathbf{1}}$

(i.e., containing e) in the left branch. A similar proof shows that the goal a is also provable from P. By Proposition 3.9, provability of e, e and a implies provability of any multiset of goals e, e, Δ and a, Δ, for every multiset of goals Δ. □

Example 3.11 Let P be the following (first-order) LO program (variables are considered externally quantified in clauses)

1. $p(x) \multimap q(y) \,\&\, r(x)$
2. $q(b) \multimap \top$
3. $r(a) \multimap \top$

where p, q, r are predicate symbols, a, b are constant symbols, and x, y are variables. A proof for the goal $p(a)$ is given on the right-hand side of Figure 3.3. Backchaining on clause 1 is carried out by considering its ground instance $p(a) \multimap q(b) \,\&\, r(a)$. □

3.3 Extensions of LO

As shown in [And92], the logical fragment underlying the language LO can be extended in order to take into consideration more powerful programming constructs. In this section we will present two different extensions of LO, which will be discussed later on. Specifically, in Section 3.3.1 we present the language LO enriched with the constant **1** and the multiplicative conjunction $\otimes$ (see also Chapter 8), and in Section 3.3.2 we present the language LO enriched with universal quantification over goals (see also Chapter 10).

3.3.1 LO with 1 and $\otimes$

In this section we consider an extension of LO where goal formulas range over the **G**-formulas of Section 3.2, the multiplicative conjunction $\otimes$ and the constant **1**, according to

the following grammar:

$$\begin{array}{lll} \mathbf{E} & ::= & \mathbf{E} \parr \mathbf{E} \mid \mathbf{E} \,\&\, \mathbf{E} \mid \mathbf{A} \mid \top \mid \bot \\ \mathbf{G} & ::= & \mathbf{E} \mid \mathbf{G} \otimes \mathbf{G} \mid \mathbf{1} \\ \mathbf{H} & ::= & \mathbf{A} \parr \ldots \parr \mathbf{A} \mid \bot \\ \mathbf{D} & ::= & \forall(\mathbf{H} \multimap \mathbf{G}) \mid \mathbf{D} \,\&\, \mathbf{D} \end{array}$$

The class **E** stands for *elementary* goals, and it is introduced in order to prevent occurrences of the synchronous connective $\otimes$ to appear inside the scope of synchronous connectives ($\parr$ and &). As shown in [And92], this is necessary in order to preserve the restriction of the *backchaining* rule to atomic contexts.

In particular, in this fragment we admit clauses of the form $H \multimap \mathbf{1}$. We name this language $\mathrm{LO}_{\mathbf{1}}$, and use the notation $P \vdash_{\mathbf{1}} \Delta$ for $\mathrm{LO}_{\mathbf{1}}$ sequents. A proof system for $\mathrm{LO}_{\mathbf{1}}$ is presented in Figure 3.4. It is obtained from the proof system of Figure 3.2 by adding a rule for $\otimes$ and a rule for the constant **1**. The meaning of the new kind of clauses $H \multimap \mathbf{1}$ is given by the following inference scheme:

$$\frac{}{P \vdash_{\mathbf{1}} \mathbf{1}}\ \mathbf{1}_r \qquad \frac{}{P \vdash_{\mathbf{1}} \widehat{H}}\ bc$$

$$\textit{provided } H \multimap \mathbf{1} \in \mathit{Gnd}(P)$$

Note that there cannot be other *resources* in the right-hand side of the lower sequent except for $\widehat{H}$. In other words, the head of the clause must match the current context *exactly* in order for rule *bc* to be fired. Thus, in contrast with $\top$, the constant **1** intuitively introduces the possibility of *counting* resources. As a result, the monotonicity property stated in Proposition 3.9, i.e., admissibility of the weakening rule, does not hold anymore for the fragment $\mathrm{LO}_{\mathbf{1}}$.

Example 3.12 Let us consider the following $\mathrm{LO}_{\mathbf{1}}$ program P:

1. $a \parr \mathit{transf} \multimap b \parr \mathit{transf}$
2. $\mathit{transf} \multimap \bot \,\&\, \mathit{check}$
3. $\mathit{check} \parr b \multimap \mathit{check}$
4. $\mathit{check} \parr c \multimap \mathit{check}$
5. $\mathit{check} \multimap \mathbf{1}$

A proof for the goal a, a, c, transf is given on the left-hand side of Figure 3.5, where, for simplicity, we have incorporated applications of the $\parr_r$, $\&_r$ and $\bot_r$ rules into backchaining steps. As we will see in Chapter 4, LO programs can be used to encode Petri nets. According to this view, every propositional symbol like a, b, c denotes one *place*, and occurrences

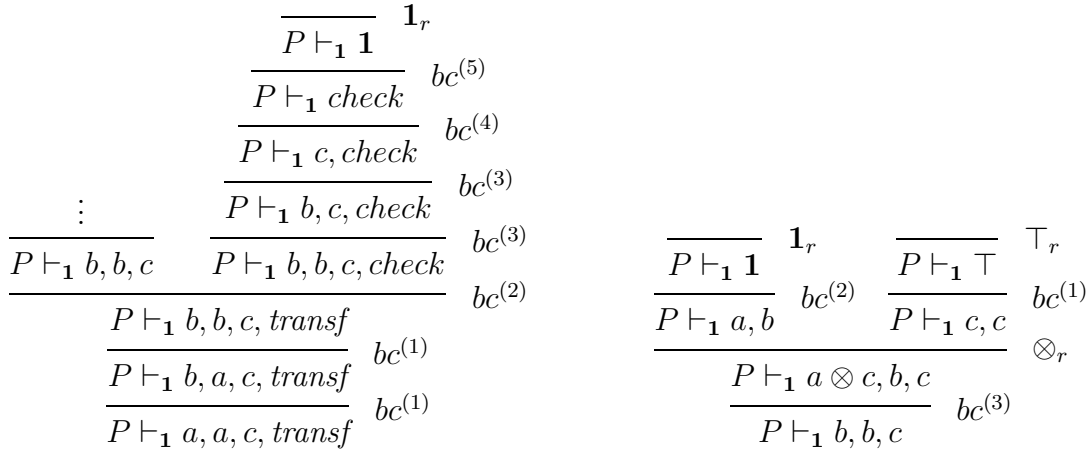

Figure 3.5: Examples of $\mathrm{LO}_{\mathbf{1}}$ proof

of symbols denote *tokens* in the corresponding place. In this view, the above program can be seen as specifying an operation *transf* which transfers *all* tokens in place a to place b. The first clause specifies the transfer of a single token from a to b, and it is supposed to be used as many times as the number of initial tokens in a. The second clause starts an auxiliary branch of the computation which checks that all tokens have been moved to b. Note that using $\mathbf{1}$ in clause 5 ensures that the check cannot succeed if there are tokens left in a. □

Example 3.13 Let us consider the following $\mathrm{LO}_{\mathbf{1}}$ program P:

1. $c \parr c \circ\!\!- \top$
2. $a \parr b \circ\!\!- \mathbf{1}$
3. $b \circ\!\!- a \otimes c$

A proof for the goal b, b, c is given on the right-hand side of Figure 3.5. □

3.3.2 LO with Universal Quantification

In this section we consider an extension of LO where the **G**-formulas of Section 3.2 are enriched with universal quantification, according to the following grammar:

$$\begin{aligned}
\mathbf{G} &::= \mathbf{G} \parr \mathbf{G} \mid \mathbf{G} \,\&\, \mathbf{G} \mid \forall x.\mathbf{G} \mid \mathbf{A} \mid \top \mid \bot \\
\mathbf{H} &::= \mathbf{A} \parr \ldots \parr \mathbf{A} \mid \bot \\
\mathbf{D} &::= \forall(\mathbf{H} \circ\!\!- \mathbf{G}) \mid \mathbf{D} \,\&\, \mathbf{D}
\end{aligned}$$

$$\frac{}{P \vdash_\Sigma \top, \Delta}\ \top_r \qquad \frac{P \vdash_\Sigma G_1, G_2, \Delta}{P \vdash_\Sigma G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash_\Sigma G_1, \Delta \quad P \vdash_\Sigma G_2, \Delta}{P \vdash_\Sigma G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash_\Sigma \Delta}{P \vdash_\Sigma \bot, \Delta}\ \bot_r \qquad \frac{P \vdash_{\Sigma,c} G[c/x], \Delta}{P \vdash_\Sigma \forall x.G, \Delta}\ \forall_r\ (c \notin \Sigma) \qquad \frac{P \vdash_\Sigma G, \mathcal{A}}{P \vdash_\Sigma \widehat{H}, \mathcal{A}}\ bc\ (H \circ\!\!- G\ \in Gnd(P))$$

Figure 3.6: A proof system for $\mathrm{LO}_\forall$

This extension is inspired by *multiset rewriting with universal quantification* [CDL+99] (see Section 5.4). The resulting language will be called $\mathrm{LO}_\forall$.

We now reformulate the proof-theoretical semantics of Section 3.2. A proof system for $\mathrm{LO}_\forall$ can be directly obtained by slightly modifying the proof system presented in Figure 3.2. Specifically, we need to make signatures explicit and add a rule for the universal quantifier. The resulting proof system is shown in Figure 3.6. Now, sequents assume the form $P \vdash_\Sigma \Delta$, where P is a program (as usual, a set of *reusable* clauses), Δ is a multiset of goals, and Σ is a signature. The signature Σ contains at least the constant, function, and predicate symbols of program P, and can dynamically grow thanks to rule $\forall_r$. The idea is that all formulas on the right-hand side of a sequent are implicitly assumed to range over Σ, i.e., only the symbols declared in Σ can appear in formulas.

Rule $\forall_r$ is responsible for signature augmentation. Every time rule $\forall_r$ is fired, a new constant c is added to the current signature, and the resulting goal is proved in the new signature. This behaviour is standard in logic programming languages [MNPS91]. Rule *bc* denotes a backchaining (resolution) step (as usual $\widehat{H}$ is the multiset consisting of the atoms in the disjunction H, with $\widehat{\bot} = \epsilon$). According to the usual concept of *uniformity*, *bc* can be executed only if the right-hand side of the current sequent consists of atomic formulas. Rules $\top_r$, $\parr_r$, $\&_r$ and $\bot_r$ are the same as in standard LO.

We can formulate the following proposition, which is analogous to Proposition 3.9.

Proposition 3.14 (Admissibility of the Weakening Rule) *Given a program P, a signature Σ, and two multisets of goals Δ, Δ' such that $\Delta \preccurlyeq \Delta'$, if $P \vdash_\Sigma \Delta$ then $P \vdash_\Sigma \Delta'$.*

Proof By simple induction on the structure of proofs. □

Example 3.15 Let Σ be a signature with a constant symbol a, a function symbol f and predicate symbols p, q, r, s. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash_{\Sigma,c} \top}\,\top_r}{P \vdash_{\Sigma,c} p(f(c)), q(f(c))}\,bc^{(4)} \quad \dfrac{\dfrac{\dfrac{}{P \vdash_{\Sigma,c} \top}\,\top_r}{P \vdash_{\Sigma,c} p(f(c)), q(f(c))}\,bc^{(4)}}{P \vdash_{\Sigma,c} p(f(c)), r(c)}\,bc^{(1)}}{P \vdash_{\Sigma,c} p(f(c)), q(f(c)) \,\&\, r(c)}\,\&_r}{P \vdash_{\Sigma,c} p(f(c))}\,bc^{(3)}}{P \vdash_{\Sigma} \forall x.p(f(x))}\,\forall_r}{P \vdash_{\Sigma} s(a)}\,bc^{(2)}$$

Figure 3.7: An example of $\mathrm{LO}_\forall$ proof

Let P be the program

1. $r(w) \circ\!\!- q(f(w))$
2. $s(z) \circ\!\!- \forall x.p(f(x))$
3. $\bot \circ\!\!- q(u) \,\&\, r(v)$
4. $p(x) \parr q(x) \circ\!\!- \top$

The goal $s(a)$ is provable from P. The corresponding proof is shown in Figure 3.7. Note that the notion of *ground instance* is now relative to the current signature. For instance, backchaining over clause 3 is possible because the corresponding signature contains the constant c (generated one level below by the $\forall_r$ rule), and therefore $\bot \circ\!\!- q(f(c)) \,\&\, r(c)$ is a valid instance of clause 3. □

Note. Subject of Part II of this thesis will be the definition of an alternative semantics for LO, based on *bottom-up* evaluation, and the investigation of the connection between provability in linear logic and verification (model checking). In particular, we will consider the fragment corresponding to propositional LO in Chapter 7, its extension with the constant $\mathbf{1}$ and the connective $\otimes$ (called $\mathrm{LO}_{\mathbf{1}}$) in Chapter 8, a formulation of LO with constraints (called $\mathrm{LO}(\mathcal{C})$) in Chapter 9, and a first-order formulation of LO with universal quantification (called $\mathrm{LO}_\forall$) in Chapter 10. In this thesis, we follow the original formulation of the language LO, as given for instance in [AP91b]. We remark that papers like [AP91a] rely on a more complex computational interpretation for LO involving *broadcast communication*. In the following, we will use the name LO and its variations as a convenient shorthand to indicate the fragment of linear logic this language is based on, even though the correspondence between the semantics we are interested in and the actual language may be questionable.

Summary of the Chapter. *In this chapter we have briefly illustrated the main ideas underlying linear logic. We have presented in more detail the language LO, its definition and a proof-theoretical, sequent-style, semantics for it. A bottom-up semantics for different fragments of LO will be the subject of Part II.*

In the next chapter we will demonstrate the potentialities of linear logic for specifying concurrent systems, by discussing its relationship with well-known formalisms like rewriting systems and Petri nets.

Chapter 4

LO, Multiset Rewriting and Petri Nets

Petri nets [Rei85] and their extensions [Jen97] have proved to be a powerful formalism for the specification and validation of concurrent systems. In this chapter, we give a brief overview of Petri nets and we discuss their relationship with linear logic and specifically with the language LO of Section 3.2. In order to study this connection, we first present the theory of *multiset rewriting systems*, which can be used to give an alternative definition for Petri nets. After showing the connection between Petri nets and multiset rewriting systems, we show how the latter can be translated into LO theories. We conclude the chapter discussing an extension of Petri nets, namely Petri nets with *transfer arcs*.

4.1 Multiset Rewriting over Atomic Formulas

As discussed in [Cer94], multisets and multiset rewriting systems can be used to give an alternative definition for the static and dynamic aspects of Petri nets. For the sake of precision, in the following we will deal with *multiset rewriting systems over atomic formulas* as defined in [Cer94], as opposed to general rewriting systems [Mes92, DP01] or AC theories [RV95]. The definition is as follows.

Definition 4.1 (Multiset Rewriting Systems over Atomic Formulas) *Let S be a finite set (alphabet). A* **multiset rewrite rule** *μ over S is a pair, written $\mu_1 \longrightarrow \mu_2$, where μ_1 and μ_2 are two multisets over S called the* **antecedent** *and the* **consequent** *of μ, respectively. A* **multiset rewriting system** *over S is a set R of multiset rewrite rules over S.*

Definition 4.2 (Configuration) *Let R be a multiset rewriting system over S. A configuration is a multiset $\mathcal{M}$ over S.*

Given a configuration $\mathcal{M}$, a rewrite rule can *fire* at $\mathcal{M}$ provided the corresponding antecedent is contained in $\mathcal{M}$. Firing is accomplished by substituting the consequent for the antecedent in $\mathcal{M}$, thus getting a new multiset $\mathcal{M}'$. This is formally stated in the following definition.

Definition 4.3 (Firing) *Let R be a multiset rewriting system over S, $\mathcal{M}$ a multiset over S, and $\mu = \mu_1 \longrightarrow \mu_2 \in R$ a rule.*

*i. We say that μ is $\mathcal{M}$-***enabled** *if $\mu_1 \preccurlyeq \mathcal{M}$;*

ii. if μ is $\mathcal{M}$-enabled, a multiset $\mathcal{M}'$ is the result of **firing** *μ at $\mathcal{M}$, written $\mathcal{M} \triangleright^{\mu} \mathcal{M}'$, if $\mathcal{M}' = (\mathcal{M} \backslash \mu_1) + \mu_2$; we write $\mathcal{M} \triangleright \mathcal{M}'$ if $\mathcal{M} \triangleright^{\mu} \mathcal{M}'$ for some μ, and we use $\triangleright^*$ to denote the reflexive and transitive closure of $\triangleright$.*

The semantics of multiset rewriting systems is formalized by the following notion of reachability set.

Definition 4.4 (Reachability Set) *Let R be a multiset rewriting system over S. Given a multiset $\mathcal{M}_I$ over S, called* **initial configuration***, we define the* **reachability set** *of R, denoted $Reach(R)$, as follows: $Reach(R) = \{\mathcal{M} \mid \mathcal{M}_I \triangleright^* \mathcal{M}\}$.*

In the next section we will present the theory of Petri nets. The mutual relations between Petri nets, multiset rewriting systems and linear logic theories will be discussed in Section 4.3.

4.2 Place/Transition Nets

In the following we shall be interested mainly in Petri nets with infinite place capacities. This is not a serious limitation, however, as arbitrary Petri nets can be encoded into Petri nets with infinite place capacities by means of the well-known operation of *complementation* [Rei85].

The traditional definition of Petri net (with infinite place capacities) is as follows.

Definition 4.5 (P/T Nets) *A* **P/T net** *is a tuple $N = \langle P, T, A, W \rangle$, where*

i. *P and T are two finite and disjoint sets, whose elements are called, respectively,* **places** *and* **transitions**;

ii. *$A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, defined by a binary relation called the* **flow relation**;

iii. *$W : A \to \mathbb{N}_0$ is the* **weight** *function.*

Definition 4.6 (Preset and Postset) *Given a Petri net $N = \langle P, T, A, W \rangle$ and $t \in T$, we define ${}^\bullet t = \{p \in P \mid (p,t) \in A\}$ and $t^\bullet = \{p \in P \mid (t,p) \in A\}$. We call ${}^\bullet t$ and $t^\bullet$, respectively, the preset and postset of the transition t.*

Definition 4.7 (Marking) *A marking for a Petri net $N = \langle P, T, A, W \rangle$ is a multiset over P, i.e., a mapping $\mathcal{M} : P \to \mathbb{N}$.*

Petri nets can be conveniently represented as labelled graphs, as in the producer/consumer example in Figure 4.1 (see also Section 4.2.1). In the graphical representation, places are drawn as circles, while transitions are drawn as squares. The flow relation is represented by arcs connecting places and transitions. Every arc is labelled with its own weight; labels with weight 1 are usually omitted for simplicity. Markings are represented by drawing as many **tokens** (i.e., black spots) into places, as the corresponding multiplicity. According to the token metaphor, **firing** of a transition t causes tokens flow from places in ${}^\bullet t$ to places in $t^\bullet$, according to the weight of the corresponding arcs. A transition t can only fire if enough tokens are present in the places in ${}^\bullet t$. This is formalized by the following definition.

Definition 4.8 (Firing) *Let $N = \langle P, T, A, W \rangle$ be a Petri net, $t \in T$ a transition, and $\mathcal{M}$ a marking for N.*

i. *We say that t is $\mathcal{M}$-***enabled** *iff $\mathcal{M}(p) \geq W(p,t)$ for all $p \in {}^\bullet t$;*

ii. *if t is $\mathcal{M}$-enabled, a marking $\mathcal{M}'$ is the result of* **firing** *t at $\mathcal{M}$, written $\mathcal{M}[t\rangle\mathcal{M}'$, iff for all $p \in P$*

$$\mathcal{M}'(p) = \begin{cases} \mathcal{M}(p) - W(p,t) \text{ iff } p \in {}^\bullet t \setminus t^\bullet \\ \mathcal{M}(p) + W(t,p) \text{ iff } p \in t^\bullet \setminus {}^\bullet t \\ \mathcal{M}(p) - W(p,t) + W(t,p) \text{ iff } p \in {}^\bullet t \cap t^\bullet \\ \mathcal{M}(p) \text{ otherwise} \end{cases}$$

we write $\mathcal{M}[\rangle\mathcal{M}'$ if $\mathcal{M}[t\rangle\mathcal{M}'$ for some t, and we use $[*\rangle$ to denote the reflexive and transitive closure of $[\rangle$.

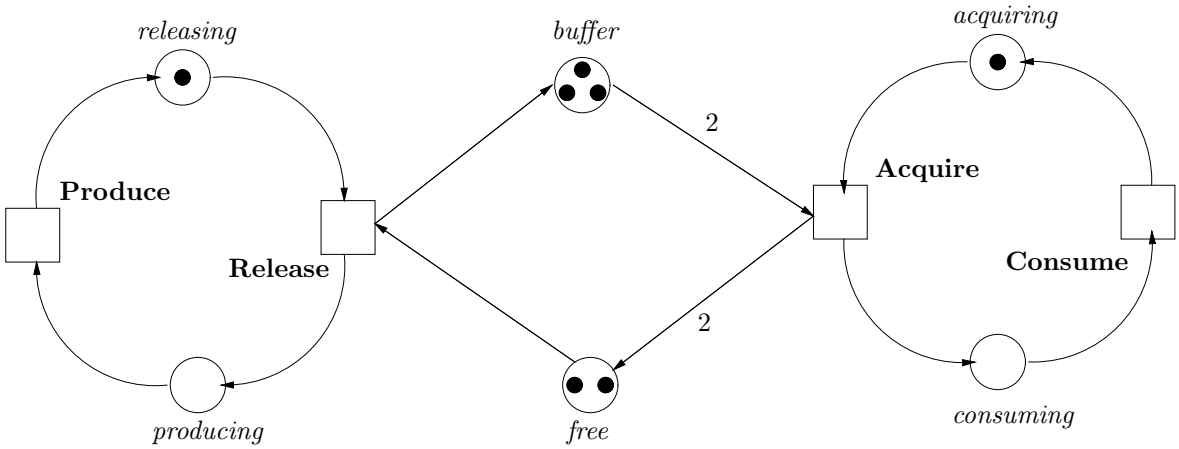

Figure 4.1: A producer/consumer net

Similarly to multiset rewriting systems, we can give the following definition for the reachability set.

Definition 4.9 (Reachability Set) *Let $N = \langle P, T, A, W \rangle$ be a Petri net. Given a marking $\mathcal{M}_I$ for N, called the* **initial marking***, we define the* **reachability set** *of N, denoted $Reach(N)$, as follows: $Reach(N) = \{\mathcal{M} \mid \mathcal{M}_I[*\rangle\mathcal{M}\}$.*

4.2.1 A Producer/Consumer Example

A simple example representing a producer/consumer net, with initial marking, is shown in Figure 4.1. The producer, represented by the left cycle, produces and releases one item at a time and puts it in a buffer. The buffer has a limited capacity of five items. This is enforced by representing the buffer by means of two places: place *buffer* contains the items released and not yet consumed, while place *free* represents the available positions in the buffer. The consumer, represented by the right cycle, acquires and consumes two items at a time, as specified by the label 2 on the arcs from *buffer* to Acquire and from Acquire to *free*.

Formally, the set of places is given by $P = \{releasing, producing, buffer, free, acquiring, consuming\}$, and the set of transitions by $T = \{$Produce,Release,Acquire,Consume$\}$. The set of arcs includes for instance $(\text{Produce}, releasing)$, $(releasing, \text{Release})$, and so on. Every arc has weight 1 except for the arcs $(buffer, \text{Acquire})$ and $(\text{Acquire}, free)$ which have weight 2. The preset of Release is given by $\{releasing, free\}$, while its postset is given by $\{producing, buffer\}$. In Figure 4.1, the following initial marking

$$\mathcal{M}_I = \{releasing, buffer, buffer, buffer, free, free, acquiring\}$$

is drawn, i.e., we have $\mathcal{M}_I(releasing) = 1$, $\mathcal{M}_I(producing) = 0$, $\mathcal{M}_I(buffer) = 3$, $\mathcal{M}_I(free) = 2$, $\mathcal{M}_I(acquiring) = 1$, $\mathcal{M}_I(consuming) = 0$. We can conveniently use the following vector notation: $\langle 1, 0, 3, 2, 1, 0\rangle$, where each position represents the number of tokens in each place, ordered as above.

The following transitions are $\mathcal{M}_I$-enabled in Figure 4.1: Release and Acquire. By firing the transition Release, we get that $\mathcal{M}_I[\text{Release}\rangle\mathcal{M}'$, with $\mathcal{M}'$ given by $\langle 0, 1, 4, 1, 1, 0\rangle$, according to the above vector notation. Similarly, by firing transition Acquire we get $\mathcal{M}_I[\text{Acquire}\rangle\mathcal{M}''$, with $\mathcal{M}''$ given by $\langle 1, 0, 1, 4, 0, 1\rangle$. As a further example, we can fire transition Consume in $\mathcal{M}''$ and get $\mathcal{M}''[\text{Consume}\rangle\mathcal{M}'''$, with $\mathcal{M}'''$ given by $\langle 1, 0, 1, 4, 1, 0\rangle$. Transition Acquire is not $\mathcal{M}'''$-enabled, because only one item is available in the buffer.

4.2.2 Place Invariants

The notion of *place invariant* is useful to perform a *static* analysis of a Petri net. For instance, given a Petri net N with set of places P, one can be interested in finding subsets of places $S \subseteq P$ such that their joint total token count does not change whatever transition may fire. This is a typical example of a property which can be statically proved by means of the place invariant analysis. Place invariants can be used to study properties of Petri nets like *liveness* or *boundedness* [Rei85].

In order to introduce the concept of place invariant, we first discuss a linear algebra representation for Petri nets.

Definition 4.10 (Matrix Representation) *Let $N = \langle P, T, A, W\rangle$ be a Petri net.*

i. Given a transition $t \in T$, we define the vector $\underline{t}$ as follows:

$$\underline{t}(p) = \begin{cases} W(t,p) \text{ iff } p \in t^\bullet \setminus {}^\bullet t \\ -W(p,t) \text{ iff } p \in {}^\bullet t \setminus t^\bullet \\ W(t,p) - W(p,t) \text{ iff } p \in {}^\bullet t \cap t^\bullet \\ 0 \text{ otherwise} \end{cases}$$

ii. we define the matrix $\underline{N} : P \times T \to \mathbb{Z}$ as follows: $\underline{N}(p,t) = \underline{t}(p)$.

Note that the above matrix representation is unambiguous (i.e., different Petri nets have different representations) only if we assume ${}^\bullet t \cap t^\bullet = \emptyset$ for every $t \in T$. Every marking of a Petri net can also be represented as a vector, as shown in Section 4.2.1. Let $\underline{\mathcal{M}}$ denote the vector corresponding to the multiset $\mathcal{M}$, i.e., $\underline{\mathcal{M}}(p) = \mathcal{M}(p)$ for every $p \in P$. The following result holds.

	Produce	Release	Acquire	Consume	$\mathcal{M}_I$
releasing	1	−1	0	0	1
producing	−1	1	0	0	0
buffer	0	1	−2	0	3
free	0	−1	2	0	2
acquiring	0	0	−1	1	1
consuming	0	0	1	−1	0

Figure 4.2: Matrix and initial marking representation

Proposition 4.11 *Let $N = \langle P, T, A, W\rangle$ be a Petri net, $t \in T$ a transition, and $\mathcal{M}$, $\mathcal{M}'$ two markings for N. If t is $\mathcal{M}$-enabled, then $\mathcal{M}[t\rangle\mathcal{M}'$ iff $\underline{\mathcal{M}'} = \underline{\mathcal{M}} + \underline{t}$.*

Proof By definitions. □

Example 4.12 Consider the producer/consumer Petri net in Figure 4.1. The corresponding matrix and initial marking vector are shown in Figure 4.2. The initial marking $\mathcal{M}_I$ is given by the vector $\langle 1, 0, 3, 2, 1, 0\rangle$, and the vector corresponding to transition Release is $\langle -1, 1, 1, -1, 0, 0\rangle$. As Release is $\mathcal{M}_I$-enabled and $\langle 1, 0, 3, 2, 1, 0\rangle + \langle -1, 1, 1, -1, 0, 0\rangle = \langle 0, 1, 4, 1, 1, 0\rangle$, we have that $\mathcal{M}_I[\text{Release}\rangle\mathcal{M}$, $\mathcal{M}$ corresponding to the vector $\langle 0, 1, 4, 1, 1, 0\rangle$. □

We have the following definition for place invariants. In the following, let $\underline{N}^T$ denote the transpose of matrix $\underline{N}$ and i^T the transpose of vector i, let $*$ be the matrix-vector or vector-vector product, and $\mathbf{0}$ a null vector.

Definition 4.13 (Place Invariants) *Let $N = \langle P, T, A, W\rangle$ be a Petri net. A place vector $i : P \to \mathbb{Z}$ is called a place invariant of N iff $\underline{N}^T * i = \mathbf{0}$.*

Place invariants are defined as solutions $\mathbf{x}$ of the linear algebraic system of equations $\underline{N}^T * \mathbf{x} = \mathbf{0}$. Let $\underline{Reach(N)}$ be the set $\{\underline{\mathcal{M}} \mid \mathcal{M} \in Reach(N)\}$. Place invariants have the following property.

Proposition 4.14 (Properties of Place Invariants [STC98]) *Let $N = \langle P, T, A, W\rangle$ be a Petri net, and $\mathcal{M}_I$ an initial marking for N. If i is a place invariant of N, then $\underline{Reach(N)} \subseteq \{\mathbf{x} \mid i^T * \mathbf{x} = i^T * \underline{\mathcal{M}_I}\}$.*

Intuitively, reachable markings must satisfy the equation $i^T * \mathbf{x} = i^T * \underline{\mathcal{M}_I}$ for every place invariant i. In other words, place invariants provide us with an over-approximation of the set of reachable markings of a given Petri net. As a special case, solutions i with elements in $\{0, 1\}$ are *characteristic* vectors of sets of places with constant token count. We also have the following result.

Proposition 4.15 *Let $N = \langle P, T, A, W\rangle$ be a Petri net, i_1 and i_2 two place invariants of N, and let $z \in \mathbb{Z}$. Then $i_1 + i_2$ and $z \cdot i_1$ are also place invariants of N.*

Proof By definition. □

Tools for automatically computing place invariants are available. For instance, specialized libraries for invariant computation are available with GreatSPN [CFGR95].

Example 4.16 Let us consider the producer/consumer net of Figure 4.1 again. The corresponding matrix is shown in Figure 4.2. Let $\mathbf{x} = \langle x_1, \ldots x_6\rangle$, $x_1, \ldots, x_6$ corresponding to places $releasing, \ldots, consuming$ in the order of Figure 4.2. Now, the associated algebraic system is

$$\begin{cases} x_1 - x_2 = 0 \\ -x_1 + x_2 + x_3 - x_4 = 0 \\ -2x_3 + 2x_4 - x_5 + x_6 = 0 \\ x_5 - x_6 = 0 \end{cases} \quad \text{which reduces to} \quad \begin{cases} x_1 = x_2 \\ x_3 = x_4 \\ x_5 = x_6 \end{cases}$$

Every place vector satisfying the above equations is a place invariant. For instance, the vectors $\langle 1, 1, 0, 0, 0, 0\rangle$, $\langle 0, 0, 1, 1, 0, 0\rangle$, and $\langle 0, 0, 0, 0, 1, 1\rangle$ are place invariants. Being characteristic vectors, we have that the total token count of the corresponding places is always constant and equals the initial marking count. Therefore we have proved the following properties: for every reachable marking $\mathcal{M}$, $\mathcal{M}(releasing) + \mathcal{M}(producing) = 1$, $\mathcal{M}(buffer) + \mathcal{M}(free) = 5$, and $\mathcal{M}(acquiring) + \mathcal{M}(consuming) = 1$. It follows, e.g., that $\mathcal{M}(buffer) \leq 5$ (the buffer cannot contain more than five elements), $\mathcal{M}(releasing) \leq 1$, and so on. According to Proposition 4.15, every linear combination of the above place vectors is still a place invariant. For instance, $\langle 1, 1, 1, 1, 1, 1\rangle$ is a place invariant, i.e., the total number of tokens in the net of Figure 4.1 is constant. □

In Part II, we will often use place invariants to optimize the performance of our verification tool (see Appendix A).

4.3 From Petri Nets to Linear Logic

After the informal discussion in the previous section, the existence of a connection between multiset rewriting systems and Petri nets should be evident. In fact, markings are exactly

multisets of places, whereas transitions can be seen as multiset rewrite rules, in accordance to the token metaphor. The following definitions make this connection explicit.

Definition 4.17 (Petri Nets as Multiset Rewriting Systems) *Given a Petri net* $N = \langle P, T, A, W \rangle$*, we define the multiset rewriting system* $R(N)$ *over* P *as follows:*

$$R(N) = \{ \sum_{p \in {}^\bullet t} W(p,t) \cdot p \longrightarrow \sum_{p \in t^\bullet} W(t,p) \cdot p \ \mid \ t \in T \}$$

We remind that $W(p,t) \cdot p$ denotes the scalar product between $W(p,t) \in \mathbb{N}$ and the singleton multiset $\{p\}$.

The following proposition states soundness and completeness of the Petri nets translation into multiset rewriting systems. The proof can be found in [Cer94].

Proposition 4.18 *Let* $N = \langle P, T, A, W \rangle$ *be a Petri net and* $\mathcal{M}, \mathcal{M}' \in \mathcal{MS}(P)$ *two markings over* P*. Then* $\mathcal{M}[*\rangle\mathcal{M}'$ *in* N *if and only if* $\mathcal{M} \triangleright^* \mathcal{M}'$ *in* $R(N)$*.*

After giving a translation from Petri nets to multiset rewriting systems, we now show how multiset rewriting systems can be specified in linear logic, and in particular in the language LO presented in Section 3.2. As a result, we also get a translation of Petri Nets into linear logic.

Definition 4.19 (Multiset Rewriting Systems as LO Theories) *Let* R *be a multiset rewriting system over a set* S*. We define the LO theory* $\mathcal{P}(R)$ *over the signature* $\Sigma = S$*, as follows:*

$$\mathcal{P}(R) = \{ H \circ\!\!- G \mid \widehat{H} \longrightarrow \widehat{G} \in R \}$$

We recall that, given a $\parr$-disjunction of atomic formulas $F = a_1 \parr \ldots \parr a_n$, $\widehat{F}$ stands for $a_1, \ldots, a_n$, while $\widehat{\bot}$ stands for the empty multiset ϵ.

Soundness and completeness of the above translation is stated in the following proposition. We omit the proof, which is a slight variation of the one presented in [Cer94] (see also [Cer95]) and can be developed using the same methodology.

Proposition 4.20 (Reachability as Provability in Propositional LO) *Let* R *be a multiset rewriting system over a set* S*,* $\mathcal{M}, \mathcal{M}' \in \mathcal{MS}(S)$ *two multisets over* S*, and* H*,* G *the (possibly empty)* $\parr$*-disjunctions of atomic formulas such that* $\widehat{H} = \mathcal{M}'$ *and* $\widehat{G} = \mathcal{M}$*. Then* $\mathcal{M} \triangleright^* \mathcal{M}'$ *in* R *if and only if* $\mathcal{P}(R), H \circ\!\!- \mathbf{1} \vdash_{\mathbf{1}} G$*.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P, D \vdash \mathbf{1}}\,\mathbf{1}_r}{P, D \vdash releasing, buffer, buffer, free, free, free, consuming}\,bc^{(D)}}{P, D \vdash releasing, buffer, buffer, buffer, buffer, free, acquiring}\,bc^{(3)}}{P, D \vdash producing, buffer, buffer, buffer, buffer, free, acquiring}\,bc^{(1)}}{P, D \vdash releasing, buffer, buffer, buffer, free, free, acquiring}\,bc^{(2)}$$

Figure 4.3: Reachability as provability in propositional LO

Example 4.21 Let us consider the Petri net for the producer/consumer example (see Figure 4.1). Its translation yields the following multiset rewriting system (one rule for each transition in $T = \{\text{Produce,Release,Acquire,Consume}\}$, respectively):

$$\{producing\} \longrightarrow \{releasing\}$$
$$\{releasing, free\} \longrightarrow \{producing, buffer\}$$
$$\{buffer, buffer, acquiring\} \longrightarrow \{free, free, consuming\}$$
$$\{consuming\} \longrightarrow \{acquiring\}$$

which corresponds to the following LO program P:

1. $producing \circ\!\!- releasing$
2. $releasing \parr free \circ\!\!- producing \parr buffer$
3. $buffer \parr buffer \parr acquiring \circ\!\!- free \parr free \parr consuming$
4. $consuming \circ\!\!- acquiring$

By firing, in the following order, the transitions Release, Produce, and Acquire, we have that $\mathcal{M}_I[*\rangle\mathcal{M}$, where $\mathcal{M}_I$ is the initial marking drawn in Figure 4.1, and $\mathcal{M}$ is the marking corresponding to the vector $\langle 1, 0, 2, 3, 0, 1\rangle$ (according to the notation of Section 4.2.1). Let D be the following $\mathrm{LO}_\mathbf{1}$ clause:

$$releasing \parr buffer \parr buffer \parr free \parr free \parr free \parr consuming \circ\!\!- \mathbf{1}$$

Figure 4.3 (where applications of the $\parr_r$ rules have been incorporated into backchaining steps, labelled as usual) shows the $\mathrm{LO}_\mathbf{1}$ proof corresponding to $\mathcal{M}_I[*\rangle\mathcal{M}$, according to Proposition 4.20. □

4.4 Petri Nets with Transfer Arcs

We conclude this chapter by discussing an extension of Petri nets, namely Petri nets with *transfer arcs*. After presenting an example, we will discuss the connection with the so-

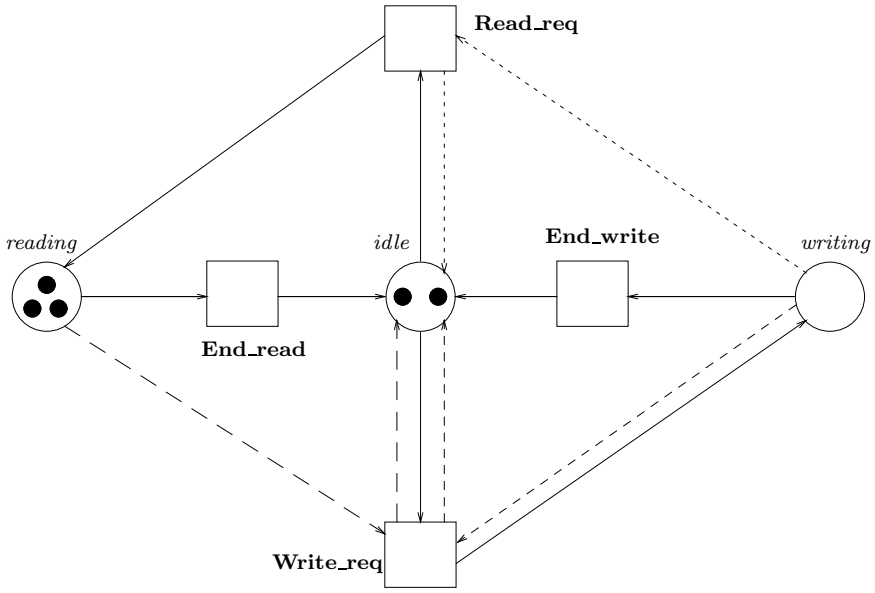

Figure 4.4: A readers/writers net

called *broadcast protocol* model [EN98]. We will give a definition and a formal semantics for broadcast protocols. We will go back to this topic in Chapter 8. In particular, in Section 8.4 we will discuss the connection between the broadcast protocol model and linear logic.

4.4.1 A Readers/Writers Example

Figure 4.4 shows a net representing a set of (identical) processes which compete for a shared resource (e.g. a memory location). Processes can be reading or writing the resource, or otherwise be idle (correspondingly, we have the places *reading*, *writing*, and *idle*). In Figure 4.4, we have drawn an initial marking in which we have three reading processes and two idle ones.

Processes are allowed to *concurrently* read the given resource, while writing must be exclusive (only one write operation and no read ones at a time). This constraint is enforced by using an *invalidation* strategy which works as follows. Every time a read request is sent (transition Read_req), the requesting process is granted the read access after all the writing processes (in this case at most one) are moved from *writing* to *idle*. In a similar way, a write request (transition Write_req) necessitates moving all processes currently in state *reading* or *writing* to state *idle*. Two further transitions, End_read and End_write, are straightforward, and allow a reading or writing process to go back to state *idle*. In Figure 4.4 we have drawn ordinary Petri net arcs in the usual way, while dotted lines (which come in pairs) represent *transfer arcs*. For instance, the pair of dotted lines from *reading* to

Write_req and from Write_req to *idle* represent the transfer of all processes in state *reading* to state *idle* which is performed upon receivement of a write request. Intuitively, the effect of a transfer arc is to move *all* tokens in a given place to another place. Transfer arcs do not affect enabling of the corresponding transition.

There is a strong connection between Petri nets with transfer arcs and *broadcast protocols*. In the next section we will present the definition of broadcast protocol and we will make the ideas presented above more precise by showing a formal semantics for this latter model.

4.4.2 Broadcast Protocols

Broadcast protocols [EN98, EFM99] are systems composed of a finite but arbitrarily large set of identical processes, which can communicate by *rendezvous* (a message is exchanged between two processes) or *broadcast* (a process sends a message to all the other processes). Typical examples of systems which can be modeled as broadcast protocols are bus-based hardware protocols, e.g. cache coherence protocols.

Definition 4.22 (Broadcast Protocol) *A broadcast protocol is a triple $B = \langle S, L, R \rangle$, where*

i. S is a finite set of **states***;*

ii. L is a finite set of **labels***, defined in the following way. Given three finite and mutually disjoint sets Σ_l, Σ_r and Σ_b, L is the union of Σ_l (the set of local labels), two sets $(\Sigma_r \times \{?\})$ and $(\Sigma_r \times \{!\})$ (the sets of input and output rendezvous labels), and two sets $(\Sigma_b \times \{??\})$ and $(\Sigma_b \times \{!!\})$ (the sets of input and output broadcast labels). We will write Σ_L for $\Sigma_l \cup \Sigma_r \cup \Sigma_b$, and we will shorten $(a, ?)$ to $a?$, and similarly for the other labels;*

iii. $R \subseteq S \times L \times S$ is a set of **transitions***. A transition (s, l, s') will be written $s \xrightarrow{l} s'$. R must satisfy the following property: for every $a \in \Sigma_b$ and state $s \in S$, there exists a state $s' \in S$ such that $s \xrightarrow{a??} s'$.*

Intuitively, the last condition on the set of transitions guarantees that every process is always willing to receive a broadcasted message. For the sake of simplicity, we will make two further assumptions on the set L: for each state s and label $a??$, there is *exactly* one state s' such that $s \xrightarrow{a??} s'$ (*determinism*); each label of the form a, $a?$ $a!$ and $a!!$ appears in exactly one transition.

Example 4.23 The readers/writers net presented in Figure 4.4 can be translated into the broadcast protocol model, as shown in Figure 4.5 (where we have not drawn the initial

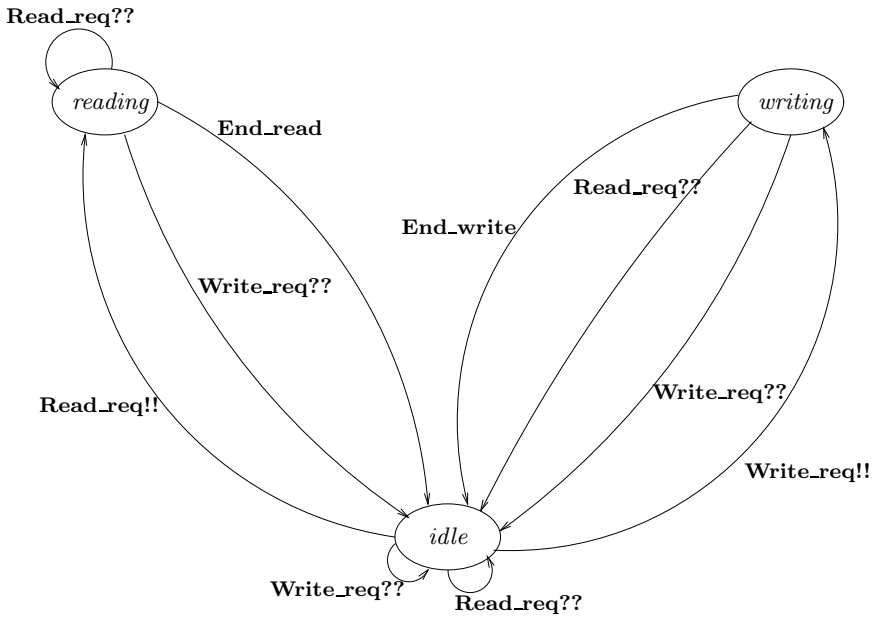

Figure 4.5: A readers/writers broadcast protocol

marking). The are three states, *reading*, *writing* and *idle*, as usual. A process willing to read broadcasts an (output) message Read_req!! to all the other processes. Upon receivement of the correspondent input message Read_req??, processes currently reading remain in state *reading*, while writing processes become *idle*. The transition for a write request is similar, with the difference that both reading and writing processes go to state *idle*. Transitions End_read and End_write are described by local labels, and their meaning is similar to ordinary Petri net transitions. No *rendezvous* transitions are needed in this example. □

We now present the semantics of broadcast protocols.

Definition 4.24 (Configuration) *A configuration for a broadcast protocol $B = \langle S, L, R\rangle$ is a multiset over S, i.e., a mapping $\mathcal{C} : S \to \mathbb{N}$.*

The notion of configuration is analogous to that of marking for Petri nets. Suppose $S = \{s_1, \ldots, s_n\}$. Then, a configuration $\mathcal{C}$ can be equivalently seen as a vector $\langle c_1, \ldots, c_n\rangle$, where $c_i = \mathcal{C}(s_i)$ (the multiplicity of s_i in $\mathcal{C}$) indicates how many processes are in state s_i [EFM99].

Definition 4.25 (Firing) *Let $B = \langle S, L, R\rangle$ be a broadcast protocol, $S = (s_1, \ldots, s_n)$, $a \in \Sigma_L$, and $\mathcal{C}$ a configuration for B. We say that $\mathcal{C}'$ is the result of firing a at $\mathcal{C}$, written $\mathcal{C} \xrightarrow{a} \mathcal{C}'$, iff*

i. $s_i \xrightarrow{a} s_j \in R$, *with* $\mathcal{C}(s_i) > 0$, *and* $\mathcal{C}' = (\mathcal{C} \setminus \{s_i\}) + \{s_j\}$;

ii. $s_i \xrightarrow{a!} s_j \in R$ *and* $s_k \xrightarrow{a?} s_l \in R$, *with* $\mathcal{C}(s_i) > 0$ *and* $\mathcal{C}(s_k) > 0$, *and* $\mathcal{C}' = (\mathcal{C} \setminus \{s_i, s_k\}) + \{s_j, s_l\}$;

iii. $s_i \xrightarrow{a!!} s_j \in R$, *with* $\mathcal{C}(s_i) > 0$, *and* $\mathcal{C}'$ *can be computed from* $\mathcal{C}$ *in the following three steps:*

$$\mathcal{C}_1 = \mathcal{C} \setminus \{s_i\}$$

$$\mathcal{C}_2(s_k) = \sum_{\{s_l \mid s_l \xrightarrow{a??} s_k\}} \mathcal{C}_1(s_l) \qquad (for \;\; k : 1, \ldots, n)$$

$$\mathcal{C}' = \mathcal{C}_2 + \{s_j\}$$

The first two cases in the previous definition are for local and rendezvous transitions, and similar to ordinary Petri net transitions. The last case is for broadcast communication. Intuitively, the sending process leaves s_i, all other processes receive the broadcast and move to their destinations, and finally the sending process reaches s_j. This is similar to a Petri net transition having pairs of *transfer* arcs for all places and in addition an ordinary pair of arcs, connecting s_i and s_j, for the sending process. From a graphical point of view, note that in Figure 4.4 pairs of transfer arcs which are *self loops* (e.g. corresponding to the arc Read_req?? for the state *reading* in Figure 4.5) are omitted. It should be evident that broadcast protocols can be seen as a special case of Petri nets with transfer arcs. The semantics of these latter nets can be given similarly.

Example 4.26 Consider the readers/writers net in Figure 4.5 and the configuration $\mathcal{C}$, corresponding to the initial marking of Figure 4.4, with 3 readers and 2 idle processes, i.e., $\mathcal{C}(reading) = 3$, $\mathcal{C}(writing) = 0$, $\mathcal{C}(idle) = 2$. Using the vector notation, $\mathcal{C}$ is given by $\langle 3, 0, 2 \rangle$. Then $\mathcal{C} \xrightarrow{\text{Read\_req}} \mathcal{C}'$, with $\mathcal{C}'$ given by $\langle 4, 0, 1 \rangle$; $\mathcal{C}' \xrightarrow{\text{Write\_req}} \mathcal{C}''$, with $\mathcal{C}''$ given by $\langle 0, 1, 4 \rangle$; $\mathcal{C}'' \xrightarrow{\text{End\_write}} \mathcal{C}'''$, with $\mathcal{C}'''$ given by $\langle 0, 0, 5 \rangle$. □

Summary of the Chapter. *In this chapter we have introduced the theories of multiset rewriting systems over atomic formulas and Petri nets, we have discussed their definition and semantics. We have analyzed the mutual relations between linear logic, multiset rewriting systems and Petri nets. We have illustrated the main ideas of this connection by means of classical examples, like producers/consumers and readers/writers. We have also mentioned the broadcast protocol model. In the next chapter we will carry on the discussion by analyzing the relations between first-order linear logic theories, first-order rewriting systems and coloured Petri nets.*

Chapter 5

First-Order LO, First-Order Multiset Rewriting and Coloured Petri Nets

According to the *token metaphor* discussed in Chapter 4, Petri nets dynamics can be described by means of tokens flowing from places to places. While the tokens considered so far are just black spots, i.e., indistinguishable from one another, a natural extension is to allow tokens to carry values, like in the so-called theory of Coloured Petri Nets (CP-nets for short) [Jen97]. The logical counterpart of CP-nets is to consider multiset rewriting over *first-order* atoms, and *first-order* LO theories, as opposed to propositional ones. In this chapter we summarize the basic concepts pertaining to CP-nets and we discuss their relationship with multiset rewriting over first-order atoms and first-order linear logic theories. We conclude by showing how enriching logic theories with universal quantification can provide a way to generate new values.

5.1 Multiset Rewriting Systems over First-Order Atomic Formulas

The theory of multiset rewriting systems presented in Section 4.1 can be extended in a straightforward way, by considering a set of rules built over first-order atomic formulas, and their ground instances. Given a denumerable set of variables $\mathcal{V}$ and a signature Σ including a set of constant and function symbols and a set of predicate symbols, atomic formulas have the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and t_i are terms in $T_\Sigma^\mathcal{V}$, for $i : 1, \ldots, n$.

Definition 5.1 (Multiset Rewriting Systems over First-Order Atoms) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. A* **multiset rewrite**

rule *μ over Σ and $\mathcal{V}$ is a pair, written $\mu_1 \longrightarrow \mu_2$, where μ_1 and μ_2 are two multisets of (non ground) atomic formulas in $A_\Sigma^\mathcal{V}$, called the* **antecedent** *and the* **consequent** *of μ, respectively. A* **multiset rewriting system** *over Σ and $\mathcal{V}$ is a set R of multiset rewrite rules over Σ and $\mathcal{V}$.*

The dynamic behaviour of a multiset rewriting system over first-order atomic formulas can be described by considering the ground instances of the set of its rewrite rules. Given a signature Σ, a possible way to achieve this is to consider a Σ-algebra D_Σ to provide a set of values (domain) and to evaluate the function and constant symbols in Σ. By analogy with Herbrand models for logic programs, in the following we assume that values are provided by T_Σ, i.e., the ground term algebra over Σ. Therefore, values are simply ground terms. The set of ground instances of a rule μ are obtained as applications of the rule itself with an evaluation σ, mapping variables in μ to terms in T_Σ. We denote by $\mu\sigma$ this application.

Definition 5.2 (Configuration) *Let R be a multiset rewriting system over Σ and $\mathcal{V}$. A configuration is a multiset $\mathcal{M}$ of ground atomic formulas in A_Σ.*

Given a configuration $\mathcal{M}$, a rewrite rule can *fire* at $\mathcal{M}$ (modulo an evaluation σ) provided the corresponding antecedent is contained in $\mathcal{M}$. Firing is accomplished by substituting the consequent for the antecedent in $\mathcal{M}$, thus getting a new multiset $\mathcal{M}'$. This is formally stated in the following definition.

Definition 5.3 (Firing) *Let R be a multiset rewriting system over Σ and $\mathcal{V}$, $\mathcal{M}$ a configuration, $\mu = \mu_1 \longrightarrow \mu_2 \in R$ a rule, and σ an evaluation for the variables in μ.*

*i. We say that μ is $\mathcal{M}$-***enabled** *with σ if $\mu_1\sigma \preccurlyeq \mathcal{M}$;*

ii. if μ is $\mathcal{M}$-enabled with σ, a multiset $\mathcal{M}'$ is the result of **firing** *μ at $\mathcal{M}$ with σ, written $\mathcal{M} \rhd^{\mu,\sigma} \mathcal{M}'$, if $\mathcal{M}' = (\mathcal{M} \setminus \mu_1\sigma) + \mu_2\sigma$; we write $\mathcal{M} \rhd \mathcal{M}'$ if $\mathcal{M} \rhd^{\mu,\sigma} \mathcal{M}'$ for some μ and σ, and we use $\rhd^*$ to denote the reflexive and transitive closure of $\rhd$.*

The usual notion of reachability set can be extended as follows.

Definition 5.4 (Reachability Set) *Let R be a multiset rewriting system over Σ and $\mathcal{V}$. Given a multiset of values $\mathcal{M}_I$, called* **initial configuration**, *we define the* **reachability set** *of R, denoted $Reach(R)$, as follows: $Reach(R) = \{\mathcal{M} \mid \mathcal{M}_I \rhd^* \mathcal{M}\}$.*

In the next section we will present the theory of coloured Petri nets. The mutual relations between coloured Petri nets, multiset rewriting systems over first-order atoms and first-order linear logic theories will be discussed in Section 5.3.

5.2 CP-nets

A (simplified) definition for Coloured Petri Nets (CP-nets) [Jen97] is as follows.

Definition 5.5 (CP-Nets) *A* **CP-net** *is a tuple* $N = \langle S, P, T, A, C, E \rangle$ *where*

i. S *is a set of* **types** *(i.e., domains enriched with operations defined over them);*

ii. P *and* T *are two finite and disjoint sets, whose elements are called, respectively,* **places** *and* **transitions***;*

iii. $A \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs, defined by a binary relation called the* **flow relation***;*

iv. $C : P \to S$ *is a function called the* **color function***, which associates every place with a type;*

v. E *is a function defined from* A *into expressions, called the* **arc expression** *function, which associates every arc with a multiset of expressions. Expressions must be of the correct type, i.e., if* $a = (p, t)$ *or* $a = (t, p)$ *then* $E(a)$ *must have the type of multisets with elements of type* $C(p)$*.*

By analogy with first-order multiset rewriting theories, the definition of S can be given by means of a (multi-sorted) algebra, defined over a signature Σ including constant and function symbols. In this case, arcs may be labelled with multisets of expressions in $T_\Sigma^{\mathcal{V}}$, i.e., the term algebra over Σ and a set of variables $\mathcal{V}$.

For convenience, in the following definition we overload the notations for presets and postsets used for ordinary Petri nets.

Definition 5.6 (Preset and Postset) *Given a CP-net* $N = \langle S, P, T, A, C, E \rangle$ *and* $t \in T$*, we define* ${}^\bullet t = \{p \in P \mid (p, t) \in A\}$ *and* $t^\bullet = \{p \in P \mid (t, p) \in A\}$*. We call* ${}^\bullet t$ *and* $t^\bullet$*, respectively, the preset and postset of the transition* t*.*

We now give the definition of binding and marking. The notion of binding is the counterpart of the notion of substitution for free variables in first-order logic.

Definition 5.7 (Bindings and Markings) *Let* $N = \langle S, P, T, A, C, E \rangle$ *be a CP-net.*

i. *A* **binding** *for a transition* $t \in T$ *is a function mapping variables in* t *(i.e., variables in* $E(a)$*, with* $a = (p, t)$ *or* $a = (t, p)$ *for some place* p*) to elements in* S*;*

ii. *a* **token element** *is a pair* (p, c)*, where* $p \in P$ *and* $c \in C(p)$*;*

iii. *a* **binding element** *is a pair* (t, b)*, where* $t \in T$ *and* b *is a binding for* t*;*

iv. *a* **marking** *is a multiset of token elements.*

Given a binding b and an expression F, in the following we will use the notation $F<b>$ to denote the application of b to F. To exemplify, suppose a CP-net is defined by means of a Σ-algebra D_Σ and F is a term in $T_\Sigma^{\mathcal{V}}$. In this case, $F<b>$ is the evaluation of F obtained by evaluating variables in F according to b, and constant and function symbols according to D_Σ. For instance, if we choose D_Σ to be the ground term algebra T_Σ, b must map variables to ground terms, and $F<b>$ is the usual term evaluation (i.e., constant and function symbols are evaluated to themselves). Evaluation can be straightforwardly extended to multisets of expressions. Finally, we will use the notation $\mathcal{M}(p)$, where $\mathcal{M}$ is a marking and p a place, to denote the multiset $\mathcal{C}$ over $C(p)$ given by $\mathcal{C}(c) = \mathcal{M}(p, c)$, for every $c \in C(p)$.

Similarly to Petri nets, CP-nets can be represented as labelled graphs. An example is shown in Figure 5.1 (see also Section 5.2.1). As usually, places are drawn as circles and transitions as squares. Every arc a is labelled with the multiset of expressions $E(a)$. Note that the multiset $E(a)$ extends the *weight function* of traditional Petri nets. In fact, in Petri nets there is only one kind of token, and we can see the weight function as giving the multiplicity of this token in the corresponding multiset. Markings can be represented by drawing as many tokens, i.e., elements in S, inside the corresponding place. The token metaphor can be extended to CP-nets. Firing of a transition is accomplished by first *binding* the free variables specified over the arcs of the transition itself to elements in S, thus getting a *binding element*, and then by letting elements flow from places to places, provided the binding element is enabled. We have the following definition.

Definition 5.8 (Firing) *Let* $N = \langle S, P, T, A, C, E \rangle$ *be a CP-net,* (t, b) *a binding element, and* $\mathcal{M}$ *a marking for* N*.*

i. *We say that* (t, b) *is* $\mathcal{M}$**-enabled** *iff* $E(p, t)<b> \preccurlyeq \mathcal{M}(p)$ *for all* $p \in {}^\bullet t$*.*

ii. *if* (t, b) *is* $\mathcal{M}$*-enabled, a marking* $\mathcal{M}'$ *is the result of* **firing** (t, b) *at* $\mathcal{M}$*, written* $\mathcal{M}[t, b\rangle\mathcal{M}'$*, iff for all* $p \in P$

$$\mathcal{M}'(p) = \begin{cases} \mathcal{M}(p) \setminus E(p, t)<b> & \text{iff } p \in {}^\bullet t \setminus t^\bullet \\ \mathcal{M}(p) + E(t, p)<b> & \text{iff } p \in t^\bullet \setminus {}^\bullet t \\ (\mathcal{M}(p) \setminus E(p, t)<b>) + E(t, p)<b> & \text{iff } p \in {}^\bullet t \cap t^\bullet \\ \mathcal{M}(p) & \text{otherwise} \end{cases}$$

we write $\mathcal{M}[\rangle\mathcal{M}'$ *if* $\mathcal{M}[t, b\rangle\mathcal{M}'$ *for some* t *and* b*, and we use* $[*\rangle$ *to denote the reflexive and transitive closure of* $[\rangle$*.*

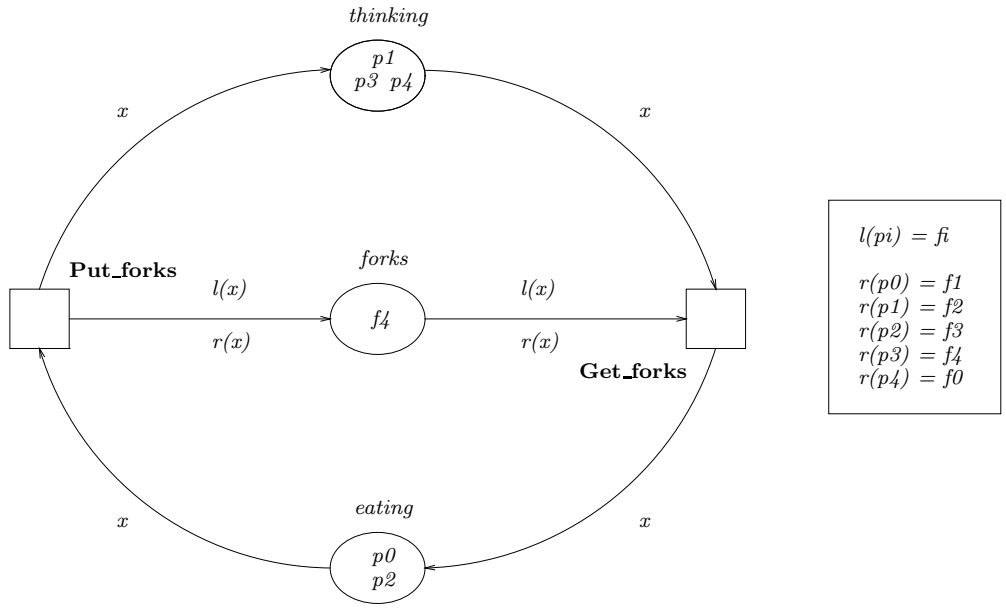

Figure 5.1: The dining philosophers net

The usual notion of reachability set can be extended as follows.

Definition 5.9 (Reachability Set) *Let $N = \langle S, P, T, A, C, E\rangle$ be a CP-net. Given a marking $\mathcal{M}_I$ for N, called the* **initial marking**, *we define the* **reachability set** *of N, denoted $Reach(N)$, as follows: $Reach(N) = \{\mathcal{M} \mid \mathcal{M}_I[*\rangle\mathcal{M}\}$.*

5.2.1 An Example: the Dining Philosophers

In Figure 5.1 we show a net representing the well-known *dining philosophers* problem. We have a set of n philosophers sitting around a (round) table and alternating periods in which they think to periods in which they eat. Each philosopher has a dish of *spaghetti* in front of him/her and two forks lie on each side of the dish (every fork is in common between two neighbouring dishes). A philosopher wishing to eat needs to take both forks, the one to the right and the one to the left of his/her dish. If either fork has already been taken, a philosopher must wait until both forks are available. Once he/she has finished eating, a philosopher puts the forks back onto the table and starts thinking again.

In Figure 5.1, we show an example for $n = 5$ philosophers, named p_i, $i : 0, \ldots, 4$. The forks are named f_i, $i : 0, \ldots, 4$, f_i being the fork to the left of philosopher p_i, and $f_{(i+1 \ mod \ 5)}$ the fork to the right. The set of places is $P = \{thinking, forks, eating\}$, containing, respectively, the set of thinking philosophers, the set of available forks, and the set of eating philosophers. We have two distinct types, namely a type S_p for philosophers and a

type S_f for forks. The functions l and r map philosophers to the respective left and right fork, as defined above. The transition Get_forks moves a philosopher x from *thinking* to *eating*, provided his/her forks $l(x)$ and $r(x)$ are available. The transition Put_forks moves a philosopher x from eating to thinking, releasing forks $l(x)$ and $r(x)$.

Initially, all philosophers are thinking and all forks are on the table. In Figure 5.1 the following marking is shown:

$$\mathcal{M} = \{(thinking, p_1), (thinking, p_3), (thinking, p_4), (forks, f_4), (eating, p_0), (eating, p_2)\},$$

i.e., we have that $\mathcal{M}(thinking) = \{p_1, p_3, p_4\}$, $\mathcal{M}(forks) = \{f_4\}$, and $\mathcal{M}(eating) = \{p_0, p_2\}$. Let b_1 be the binding such that $b_1(x) = p_0$. The binding element $(\text{Put\_forks}, b_1)$ is enabled in $\mathcal{M}$. We have that $\mathcal{M}[\text{Put\_forks}, b_1\rangle\mathcal{M}'$, where $\mathcal{M}'(thinking) = \{p_0, p_1, p_3, p_4\}$, $\mathcal{M}'(forks) = \{f_0, f_1, f_4\}$, and $\mathcal{M}'(eating) = \{p_2\}$. As the reader can verify, the transition Get_forks is not enabled in $\mathcal{M}$ for any binding b, because only one fork is available (at most two philosophers can eat at the same time). As another example, let b_2 be the binding such that $b_2(x) = p_4$. The binding element $(\text{Get\_forks}, b_2)$ is enabled in $\mathcal{M}'$, and $\mathcal{M}'[\text{Get\_forks}, b_2\rangle\mathcal{M}''$, where $\mathcal{M}''(thinking) = \{p_0, p_1, p_3\}$, $\mathcal{M}''(forks) = \{f_1\}$, and $\mathcal{M}''(eating) = \{p_2, p_4\}$.

5.3 From CP-nets to Linear Logic

In this section we will discuss the connections between CP-nets, first-order multiset rewriting and first-order linear logic theories. Basically, we can use the same ingredients as in Section 4.3, where we formally stated a similar connection between ordinary Petri nets, rewriting and propositional linear logic theories. However, the correspondence between CP-nets as defined in [Jen97] and first-order multiset rewriting (or linear logic theories) is not perfect. In particular, our linear logic theories as presented so far are untyped (though a type theory could certainly be introduced), and furthermore in [Jen97] CP-nets are provided with a rich ML-like language for defining operations on type values. For these reasons, and having in mind the purpose of the present thesis work, we will feel free to keep the tone of this discussion somewhat informal. In particular, we will demonstrate the connection between CP-nets and multiset rewriting over first-order atoms by directly showing the encoding of the dining philosophers example of Section 5.2.1.

Example 5.10 Consider the dining philosopher CP-net of Figure 5.1. The idea is to take the set of places as the set of predicate symbols of the corresponding theory. We have three places which we can encode by means of the predicate symbols *thinking*, *forks* and *eating*. We need at least two function symbols l and r to encode the left and right fork functions, and we can also use two further term constructors p and f to simulate the two types for philosophers and forks, respectively. We can use natural numbers to encode philosopher

indices. To exemplify, the terms $p(0), \ldots, p(n)$ will denote the set of n philosophers, and $f(0), \ldots, f(n)$ the set of forks. The initial marking presented in Figure 5.1 corresponds to the following multiset:

$$\mathcal{M} = \{thinking(p(1)), thinking(p(3)), thinking(p(4)), forks(f(4)), eating(p(0)), eating(p(2))\}.$$

The transition Get_forks can be encoded by means of the following rewrite rule, where x is a variable:

$$\{thinking(p(x)), forks(l(p(x))), forks(r(p(x)))\} \longrightarrow \{eating(p(x))\}$$

and, similarly, the following rewrite rule encodes the transition Put_forks:

$$\{eating(p(x))\} \longrightarrow \{thinking(p(x)), forks(l(p(x))), forks(r(p(x)))\}$$

Now, we need to define rules to implement the l and r functions, e.g. using the following encoding :

$$\{forks(l(p(x)))\} \longrightarrow \{forks(f(x))\}$$
$$\{forks(r(p(x)))\} \longrightarrow \{forks(f(x+1 \ mod \ n))\}$$
$$\{forks(f(x))\} \longrightarrow \{forks(l(p(x)))\}$$
$$\{forks(f(x+1 \ mod \ n))\} \longrightarrow \{forks(r(p(x)))\}$$

where the *mod* operator is defined via a suitable arithmetic theory. A simple alternative is to consider a set of rules, one for each possible value of x, as suggested in Figure 5.1 (in this case the set of values is finite). In general, we need to enrich our rewriting theory with clauses for all the operations defined in the original CP-net. □

The connection between multiset rewriting systems over first-order atomic formulas and first-order LO theories can be stated by analogy with the propositional case of Section 4.3.

Definition 5.11 (First-Order Multiset Rewriting Systems as LO Theories) *Let R be a multiset rewriting system over Σ and $\mathcal{V}$. We define the LO theory $\mathcal{P}(R)$ (over Σ and $\mathcal{V}$), as follows:*

$$\mathcal{P}(R) = \{\forall\,(H \circ\!\!- G) \mid \widehat{H} \longrightarrow \widehat{G} \in R\}$$

As usual, the notation $\forall\,(H \circ\!\!- G)$ stands for the universal quantification of clause $H \circ\!\!- G$ over its free variables, whereas $\widehat{a_1 \parr \ldots \parr a_n}$ stands for $a_1, \ldots, a_n$, with $\widehat{\bot} = \epsilon$. An analogous soundness and completeness result holds.

Proposition 5.12 (Reachability as Provability in First-Order LO) *Let R be a multiset rewriting system over Σ and $\mathcal{V}$, $\mathcal{M}, \mathcal{M}' \in \mathcal{MS}(A_\Sigma)$ two configurations, and H, G the (possibly empty) $\parr$-disjunctions of ground atomic formulas such that $\widehat{H} = \mathcal{M}'$ and $\widehat{G} = \mathcal{M}$. Then $\mathcal{M} \triangleright^* \mathcal{M}'$ in R if and only if $\mathcal{P}(R), H \circ\!\!- \mathbf{1} \vdash_{\mathbf{1}} G$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P, D \vdash \mathbf{1}}\,\mathbf{1}_r}{P, D \vdash thinking(p(0), p(1), p(3)), forks(f(1)), eating(p(2), p(4))}\,bc^{(D)}}{P, D \vdash thinking(p(0), p(1), p(3), p(4)), forks(r(p(4)), f(1), l(p(4))), eating(p(2))}\,bc^{(1)}}{P, D \vdash thinking(p(0), p(1), p(3), p(4)), forks(f(0), f(1), l(p(4))), eating(p(2))}\,bc^{(6)}}{P, D \vdash thinking(p(0), p(1), p(3), p(4)), forks(f(0), f(1), f(4)), eating(p(2))}\,bc^{(5)}}{P, D \vdash thinking(p(0), p(1), p(3), p(4)), forks(f(0), r(p(0)), f(4)), eating(p(2))}\,bc^{(4)}}{P, D \vdash thinking(p(0), p(1), p(3), p(4)), forks(l(p(0)), r(p(0)), f(4)), eating(p(2))}\,bc^{(3)}}{P, D \vdash thinking(p(1), p(3), p(4)), forks(f(4)), eating(p(0), p(2))}\,bc^{(2)}$$

Figure 5.2: Reachability as provability in first-order LO

Example 5.13 We can collect the multiset rewriting rules of Example 5.10 into the following first-order LO program P (where x is implicitly universally quantified):

1. $thinking(p(x)) \parr forks(l(p(x))) \parr forks(r(p(x))) \circ\!\!- eating(p(x))$
2. $eating(p(x)) \circ\!\!- thinking(p(x)) \parr forks(l(p(x))) \parr forks(r(p(x)))$
3. $forks(l(p(x))) \circ\!\!- forks(f(x))$
4. $forks(r(p(x))) \circ\!\!- forks(f(x + 1 \ mod \ n))$
5. $forks(f(x)) \circ\!\!- forks(l(p(x)))$
6. $forks(f(x + 1 \ mod \ n)) \circ\!\!- forks(r(p(x)))$

Let D be the following $\mathrm{LO}_{\mathbf{1}}$ clause:

$$thinking(p(0)) \parr thinking(p(1)) \parr thinking(p(3)) \parr forks(f(1)) \parr eating(p(2)) \parr eating(p(4)) \circ\!\!- \mathbf{1}.$$

The derivation corresponding to firing $\mathcal{M}[\mathrm{Put\_forks}, b_1\rangle\mathcal{M}'[\mathrm{Get\_forks}, b_2\rangle\mathcal{M}''$ of Section 5.2.1 (according to Proposition 5.12) is shown in Figure 5.2. We have used the following convention: a formula like $thinking(p_1, \ldots, p_n)$ is a shorthand for the multiset $thinking(p_1), \ldots, thinking(p_n)$, and similarly for $forks(\ldots)$ and $eating(\ldots)$. As usual, applications of the $\parr_r$ rules have been incorporated into backchaining steps. □

5.4 Multiset Rewriting with Universal Quantification

We conclude this chapter by discussing an extension of first-order multiset rewriting with *universal quantification*. In the *proof as computation* interpretation of logic programs, universal quantification is a logical operator which provides a way to generate new values. From a logical perspective, this view of universal quantification is based on its proof-theoretical semantics in intuitionistic (and also linear) logic [MNPS91]. We will define first-order multiset rewriting with universal quantification taking inspiration from [CDL$^+$99], where a similar logic fragment, called MSR, is defined. In [CDL$^+$99], MSR is used for the specification and analysis of *security protocols*, a topic we will discuss in Chapter 11.

Given the direct relationship between (first-order) multiset rewriting and (first-order) linear logic (in particular the fragment LO, as shown in Section 4.3), it should be evident that multiset rewriting with universal quantification is the counterpart of LO with universal quantification (see Section 3.3.2). As an aside, we remark that in [CDL$^+$99] the logic MSR is compared to a fragment of linear logic which turns out to be *dual* with respect to ours, and therefore *existential* quantification is used in place of universal quantification.

The following definitions extend those presented in Section 5.1. As new values (constants) can be generated, multiset rewriting systems are defined on dynamically growing signatures, which we must keep trace of. Given a signature (with predicates) Σ and a set of new constants C (not appearing in Σ), we will use the notation $\Sigma \cup C$ to denote the addition of the constants in C to the set of constants in Σ.

Definition 5.14 (Quantified Multiset Rewriting Systems) *Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. A* **quantified multiset rewrite rule** *μ over Σ and $\mathcal{V}$ is a pair, written $\mu_1 \longrightarrow \forall \mathbf{x}.\mu_2$, where μ_1 and μ_2 are two multisets of (non ground) atomic formulas in $A^{\mathcal{V}}_{\Sigma}$, called the* **antecedent** *and the* **consequent** *of μ, respectively, and $\mathbf{x}$ is a vector of variables $\langle x_1, \ldots, x_k \rangle$, with $x_i \in \mathcal{V}$ for $i : 1, \ldots, k$. A* **quantified multiset rewriting system** *is a set R of multiset rewrite rules over Σ and $\mathcal{V}$.*

The dynamic behaviour of a quantified multiset rewriting system can be described by considering the ground instances of the set of its rewrite rules, which can be obtained as explained in the following. First of all, note that the notion of instance (evaluation of terms to base values) is *relative* to the current signature, i.e., if the signature is augmented with a new constant c, we must provide a new value to evaluate c. By analogy with Section 5.1, given a signature Σ_1 we assume that the set of values are provided by T_{Σ_1}, i.e., the ground term algebra over Σ_1. We need to extend the notion of configuration as follows.

Definition 5.15 (Configuration) *Let R be a quantified multiset rewriting system over*

Σ and $\mathcal{V}$. A configuration is a pair, written $\langle \mathcal{M}, \Sigma_1 \rangle$, where Σ_1 is a signature such that $\Sigma \subseteq \Sigma_1$ and $\mathcal{M} \in \mathcal{MS}(T_{\Sigma_1})$ is a multiset of terms.

Instantiation of a rule works as follows. Given a rule $\mu = \mu_1 \longrightarrow \forall \mathbf{x}.\mu_2$, with $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$, and a configuration $\langle \mathcal{M}, \Sigma_1 \rangle$, we first substitute new constants $c_1, \ldots, c_k$ (not appearing in Σ_1) for the bound variables $\mathbf{x}$. We then substitute values (terms in T_{Σ_1}) for the free variables in μ. Firing will yield a new configuration $\langle \mathcal{M}', \Sigma_2 \rangle$, with $\Sigma_2 = \Sigma_1 \cup \{c_1, \ldots, c_k\}$. Given an evaluation σ of the *free* variables in μ, we denote by $\mu\sigma$ the application of σ to μ. As explained above, the notion of evaluation is relative to a signature. We have the following definition.

Definition 5.16 (Firing) *Let R be a quantified multiset rewriting system over Σ and $\mathcal{V}$, $\langle \mathcal{M}, \Sigma_1 \rangle$ a configuration, $\mu = \mu_1 \longrightarrow \forall \mathbf{x}.\mu_2 \in R$ a rule, with $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$, and σ an evaluation mapping free variables in μ to terms in T_{Σ_1}.*

*i. We say that μ is $\langle \mathcal{M}, \Sigma_1 \rangle$-***enabled** *with σ if $\mu_1\sigma \preccurlyeq \mathcal{M}$;*

ii. if μ is $\langle \mathcal{M}, \Sigma_1 \rangle$-enabled with σ, a multiset $\langle \mathcal{M}', \Sigma_2 \rangle$ is the result of **firing** *μ at $\langle \mathcal{M}, \Sigma_1 \rangle$ with σ, written $\langle \mathcal{M}, \Sigma_1 \rangle \rhd^{\mu,\sigma} \langle \mathcal{M}', \Sigma_2 \rangle$, if $\mathbf{c} = \langle c_1, \ldots, c_k \rangle$ is a vector of new constants (not appearing in Σ_1), $\Sigma_2 = \Sigma_1 \cup \{c_1, \ldots, c_k\}$, and $\mathcal{M}' = (\mathcal{M} \backslash \mu_1\sigma) + (\mu_2[\mathbf{c}/\mathbf{x}])\sigma$; we write $\langle \mathcal{M}, \Sigma_1 \rangle \rhd \langle \mathcal{M}', \Sigma_2 \rangle$ if $\langle \mathcal{M}, \Sigma_1 \rangle \rhd^{\mu,\sigma} \langle \mathcal{M}', \Sigma_2 \rangle$ for some μ and σ, and we use $\rhd^*$ to denote the reflexive and transitive closure of $\rhd$.*

Example 5.17 Let Σ be a signature with two constant symbols a and b, one function symbol f and two predicate symbols p, q. Let $\mathcal{V}$ be a denumerable set of variables and $w, x, y \in \mathcal{V}$. Let μ be the rule

$$\{p(x), q(f(y))\} \longrightarrow \forall w.\{p(f(x)), q(y), q(w)\}$$

and $\mathcal{M} = \{p(f(a)), p(b), q(f(b))\}$. Then we can fire rule μ at $\langle \mathcal{M}, \Sigma \rangle$ with the evaluation σ s.t. $\sigma(x) = \sigma(y) = b$. Let c be a new constant, and $\Sigma_1 = \Sigma \cup \{c\}$. Then we get $\langle \mathcal{M}, \Sigma \rangle \rhd^{\mu,\sigma} \langle \mathcal{M}', \Sigma_1 \rangle$, where $\mathcal{M}' = \{p(f(a)), p(f(b)), q(b), q(c)\}$. □

The connection between first-order multiset rewriting systems and first-order LO theories can be extended to quantified multiset rewriting systems and $\mathrm{LO}_\forall$ theories, as follows.

Definition 5.18 (Quantified Multiset Rewriting Systems as $\mathrm{LO}_\forall$ Theories) *Let R be a quantified multiset rewriting system over Σ and $\mathcal{V}$. We define the $LO_\forall$ theory $\mathcal{P}(R)$ (over Σ and $\mathcal{V}$), as follows:*

$$\mathcal{P}(R) = \{\forall\,(H \circ\!\!- \forall \mathbf{x}.G) \mid \widehat{H} \longrightarrow \forall \mathbf{x}.\widehat{G} \in R\}$$

$$\dfrac{\dfrac{\dfrac{}{P, D \vdash_{\Sigma,c} \mathbf{1}}\ \mathbf{1}_r}{P, D \vdash_{\Sigma\, c} p(f(a)), p(f(b)), q(b), q(c)}\ bc^{(D)}}{P, D \vdash_{\Sigma} p(f(a)), p(b), q(f(b))}\ bc^{(1)}$$

Figure 5.3: Reachability as provability in $\mathrm{LO}_\forall$

A soundness and completeness result holds. Note that, in order to state the following proposition, we need to consider provability in $\mathrm{LO}_\forall$ augmented with the rule $\mathbf{1}_r$ (see Sections 3.3.1 and 3.3.2).

Proposition 5.19 (Reachability as Provability in $\mathrm{LO}_\forall$) *Let R be a quantified multiset rewriting system over Σ and $\mathcal{V}$, C_1 and C_2 two configurations such that $C_1 = \langle \mathcal{M}, \Sigma_1 \rangle$ and $C_2 = \langle \mathcal{M}', \Sigma_2 \rangle$, and H, G the (possibly empty) $\parr$-disjunctions of ground atomic formulas such that $\widehat{H} = \mathcal{M}'$ and $\widehat{G} = \mathcal{M}$. Then $C_1 \triangleright^* C_2$ in R if and only if $\mathcal{P}(R), H \circ\!\!- \mathbf{1} \vdash_{\Sigma_1} G$.*

Example 5.20 Let Σ be the signature of Example 5.17. The $\mathrm{LO}_\forall$ program P corresponding to the rule given in Example 5.17 consists of the following clause:

$$1.\ \ p(x) \parr q(f(y)) \circ\!\!- \forall w.(p(f(x)) \parr q(y) \parr q(w))$$

Let D be the clause $p(f(a)) \parr p(f(b)) \parr q(b) \parr q(c) \circ\!\!- \mathbf{1}$. The derivation corresponding to $\langle \mathcal{M}, \Sigma \rangle \triangleright^{\mu,\sigma} \langle \mathcal{M}', \Sigma_1 \rangle$ (see Example 5.17), with the usual conventions, is shown in Figure 5.3. □

Finally, the usual notion of reachability set can be extended as follows.

Definition 5.21 (Reachability Set) *Let R be a quantified multiset rewriting system over Σ and $\mathcal{V}$. Given a configuration $\langle \mathcal{M}_I, \Sigma \rangle$, called* **initial configuration***, we define the* **reachability set** *of R, denoted $Reach(R)$, as follows: $Reach(R) = \{\langle \mathcal{M}, \Sigma_1 \rangle \mid \langle \mathcal{M}_I, \Sigma \rangle \triangleright^* \langle \mathcal{M}, \Sigma_1 \rangle\}$.*

Summary of the Chapter. *In this chapter we have presented the theories of multiset rewriting over first-order atomic formulas and coloured Petri nets, an extension of the Petri net model in which tokens can carry along values. We have illustrated the connection between these formalisms and first-order linear logic theories. We have also formulated the results for an extension of multiset rewriting systems with universal quantification, and the corresponding fragment $LO_\forall$.*

After substantiating our view of linear logic as a formalism for concurrency in Part I, subject of Part II will now be the definition of a bottom-up semantics for linear logic programs, which will be used as a tool for model checking of parameterized and infinite-state systems. First of all, in the next chapter we will explain in detail our approach to verification and the connection with linear logic semantics.

Part II

Model Checking for Linear Logic Specifications

Chapter 6

A Backward Approach to Model Checking

In Part I we have presented the basics of linear logic, specifically the language LO [AP91b] and some proper extensions, and we have analyzed the mutual relations between linear logic and different formalisms for the specification of concurrent systems, like Petri nets and multiset rewriting systems. This chapter is intended to provide the connection between linear logic and the problem of verification for *parameterized* concurrent systems, which is the focal point of this thesis work. Verification can be performed via an algorithmic procedure which is based on a *backward* evaluation strategy [ACJT96] and has strong similarities with symbolic model checking techniques [KMM+97b]. The purpose of this chapter is to formally motivate this approach and to discuss its applicability.

In the case of ordinary Petri nets, the verification algorithm proposed in [ACJT96] has a counterpart in the so-called Karp and Miller's *coverability graph* computation [KM69]. The contribution of this thesis will be to extend this view towards richer specification languages, like LO. In particular, propositional LO contains a subset which can be used to encode Petri nets, as demonstrated in Section 4.3. Furthermore, in Section 5.3 we have shown that first-order formulations of LO have a direct connection with high-level extensions of nets, like coloured Petri nets. In the following chapters, we will discuss a verification procedure for specifications written in different fragments of linear logic, which we will apply to solve problems in application domains like concurrency theory or security protocols. We will also isolate interesting fragments for which termination of the verification procedure is guaranteed.

This chapter is structured as follows. We first present the problem of verification using a rather general formulation on transition systems. In particular, we focus on *reachability* problems. Then, we describe a backward verification procedure and the class of problems

this procedure is able to solve. We discuss the requirements which are needed to ensure effectiveness and a general theory providing sufficient conditions for termination. We clarify the connection between the verification procedure and *bottom-up* semantics of linear logic. Much of the background material for this chapter is a partial and free re-elaboration of [FS01, AJ01b].

6.1 Transition Systems and Verification

We can formulate the problem of verification for arbitrary transition systems. We need the following definitions.

Definition 6.1 (Transition System) *A transition system is a pair $T = \langle S, R \rangle$, where S is a set of* **states** *and $R \subseteq S \times S$ is a set of* **transitions**. *A transition (s, s') is usually noted $s \longrightarrow s'$, and the reflexive and transitive closure of the $\longrightarrow$ relation is denoted $\longrightarrow^*$. If $s_1 \longrightarrow^* s_2$, we say that there is an* **execution trace** *from state s_1 to state s_2.*

A transition system may have additional structure like, e.g., labels on transitions or initial states. As an example, the operational semantics of Petri nets, multiset rewriting systems and broadcast protocols (see Chapters 4 and 5) induces a transition system. For instance, the *firing* relation of Petri nets is clearly an example of transition relation on *markings*.

Definition 6.2 (Predecessor and Successor) *Given a transition system $T = \langle S, R \rangle$, and $W \subseteq S$, we define the predecessor and successor operators, respectively, as follows: $Pre(W) = \{s \in S \mid s \longrightarrow w,\ w \in W\}$ and $Post(W) = \{s \in S \mid w \longrightarrow s,\ w \in W\}$. We denote the reflexive and transitive closure of Pre and $Post$ by Pre^* and $Post^*$.*

We now discuss the concept of system verification, with particular emphasis on problems involving the notion of *reachability*. Informally, reachability properties can be described as follows. We are given a set of initial states and a set of final states of a given (transition) system T, and the problem is to verify whether some final state can be reached with an execution trace starting from an initial state. Verification of *safety properties* is a typical example of problem which falls into this class. Safety problems can be described in temporal logic [MP95] as follows. Given a property φ, verifying whether φ holds for every reachable state amounts to checking whether $AG\varphi$ (*always globally* φ) holds. The previous expression is logically equivalent to $\neg EF\neg\varphi$ (*not exists finally not* φ), therefore the problem is equivalent to verifying whether $EF\psi$ holds, where ψ represents the *negation* of the safety property under consideration. To exemplify, we may want to verify whether an *undesired* state (e.g., in concurrency theory, a state in which a *deadlock* or a violation of *mutual exclusion* occur) is reachable starting from a valid initial configuration.

Definition 6.3 (Reachability Problem) *Let $T = \langle S, R \rangle$ be a transition system, and $s_1, s_2 \in S$ two states. We say that s_2 is reachable from s_1 if $s_1 \longrightarrow^* s_2$. Given a set $I \subseteq S$ of initial states and a set $F \subseteq S$ of final states, the reachability problem consists in determining if $i \longrightarrow^* f$ for some $i \in I$ and $f \in F$.*

Different approaches can be used to tackle the reachability problem. An approach which is suitable for automation is to systematically and exhaustively search for all execution traces from some initial state to some final state. Clearly, the number of execution traces in general can be infinite, therefore the problem of termination must be addressed. A distinction can be made depending on the search strategy. Classical examples of strategies include *forward* and *backward* search. Mixed strategies are also possible [Hol88]. In the *forward* approach, the search starts from the set of initial states, while in the backward approach the search starts from the set of final states. This is explained by the following proposition.

Proposition 6.4 (Forward and Backward Search) *Let $T = \langle S, R \rangle$ be a transition system, $I \subseteq S$ a set of initial states and $F \subseteq S$ a set of final states. Then the reachability problem amounts to checking whether $Post^*(I) \cap F \neq \emptyset$* (**forward search**)*, or, equivalently, whether $Pre^*(F) \cap I \neq \emptyset$* (**backward search**)*.*

Proof Immediate by definitions. □

The previous proposition suggests a way to solve the reachability problem. In the case of *forward* search, the idea is to compute the $Post^*$ operator as transitive closure of the $Post$ operator, starting from the set I. Similarly, *backward* search works by computing Pre^* as transitive closure of Pre, starting from F. At this level, the forward and backward search strategies are perfectly symmetrical. Let us fix, e.g., a backward search strategy. Intuitively, the idea is to start from F and generate, in a *breadth-first* search style, the set of states from which a state in F can be reached, using an *iterative* procedure. At the first step, we compute the set of states from which a state in F can be reached with an execution trace of length one or zero. In general, at step j we compute the set of states from which F can be reached with an execution trace of length at most j. If the procedure converges, i.e., the sets computed at steps j and $j + 1$ coincide, one checks whether the intersection with the set of initial states I is empty or not.

This (very abstract) verification procedure is described in Figure 6.1. Clearly, at least two important issues must be addressed. First, the sets computed at each step in general are infinite. Therefore, in order for the procedure to work, we need a symbolic way (i.e., an *assertional language*) to represent infinite sets of states, and we need a version of the Pre operator working on symbolic sets of states. In this way, we will be guaranteed that the computation of each step of the verification procedure is *effective*. Second, we need

```
Procedure Reachability
input T = (S, R): a transition system
        I: a set of initial states; F: a set of final states
output Pre*(F) ∩ I = ∅
begin
    W := F; V := ∅
    while W ⊈ V do
        if W ∩ I ≠ ∅ then return reachable
        else V := W
             W := W ∪ Pre(W)
        endif
    endwhile
    return unrechable
end
```

Figure 6.1: An abstract verification procedure for reachability

(sufficient) conditions ensuring termination of the verification procedure. The first issue will be addressed in Section 6.1.1, where we introduce the notion of *structured transition system*, while the problem of termination will be discussed in Section 6.2.

6.1.1 Structured Transition Systems

In order to refine the abstract verification procedure for solving the reachability problem of Figure 6.1, in this section we discuss transition systems which satisfy additional hypotheses. In particular, we consider transition systems with additional structure, namely an ordering relation, on the set of states. We have the following definition.

Definition 6.5 (Structured Transition System) *A structured transition system is a tuple $T = \langle S, R, \leqslant \rangle$, where $\langle S, R \rangle$ is a transition system and $\leqslant$ is a quasi-order on S.*

We remind that a quasi-order is a reflexive and transitive binary relation.

Example 6.6 A *multiset rewriting system* (see Section 4.1) clearly induces a transition system in which states are multisets over a set S. It also induces a structured transition system where the quasi-order is given by the sub-multiset relation $\preccurlyeq$. Thanks to the results of Section 4.3, Petri nets are equivalent to multiset rewriting systems and therefore can be seen in turn as structured transition systems. □

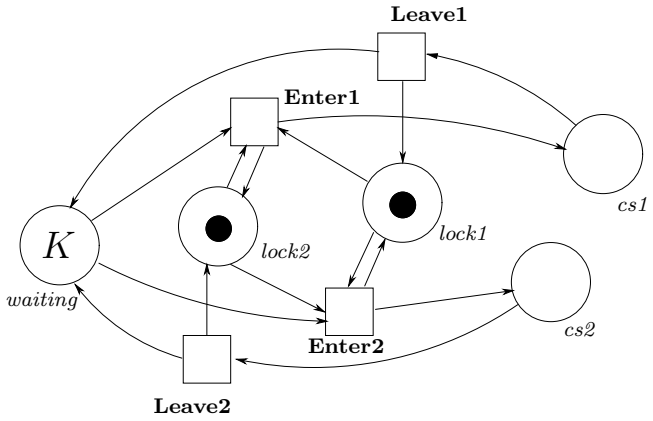

Figure 6.2: Illustrating the covering problem: a Petri net for mutual exclusion

Now, an interesting instance of the reachability problem of Section 6.1 is the following, which goes under the name of *covering problem*, and consists in determining if starting from a state s it is possible to *cover* t, i.e., to reach a state t' such that $t \leqslant t'$.

Definition 6.7 (Covering Problem) *Let $T = \langle S, R, \leqslant \rangle$ be a structured transition system. Given a set $I \subseteq S$ of initial states and a set $F \subseteq S$ of final states, the covering problem consists in determining if there exist $i \in I$, $f \in F$, and $f' \in S$, such that $i \longrightarrow^* f'$ and $f \leqslant f'$.*

The covering problem is sometimes referred to as *control-state reachability problem* [FS01]. The covering problem is particularly meaningful because many *safety* problems in concurrency theory, e.g. *mutual exclusion*, can be described in this way.

Example 6.8 Let us consider the Petri net in Figure 6.2, representing a monitor for a parameterized system with two mutually exclusive critical sections, $cs1$ and $cs2$. The Petri net is parametric in the number of processes. Initially, we have K processes in place $waiting$, one token in $lock1$ and one token in $lock2$. In other words, the initial marking, drawn in Figure 6.2, is $\mathcal{M}_I = K \cdot \{waiting\} + \{lock1, lock2\}$. A process wishing to enter $cs1$ (transition Enter1) must test for the presence of processes in $cs2$ using $lock2$, and locks cs_1 using $lock1$. Transition Enter2 is specular. Processes leave the critical section using transitions Leave1 and Leave2. Mutual exclusion requires that at most one process can be executing inside either critical section at the same time, and that the two critical sections are mutually exclusive. Accordingly, a marking violating mutual exclusion has the form $\{cs1, cs1\} + \mathcal{M}$, $\{cs1, cs2\} + \mathcal{M}$, or $\{cs2, cs2\} + \mathcal{M}$, where $\mathcal{M}$ is any marking. In other words, a configuration is *unsafe* if there are *at least* two processes accessing one of the two critical sections at the same time. Therefore, mutual exclusion is violated if there exists a valid initial state from which it is possible to *cover* a state having the form $\{cs1, cs1\}$,

$\{cs1, cs2\}$, or $\{cs2, cs2\}$. Verifying the problem of mutual exclusion amounts to solving a covering problem in which the set of initial states are the legal initial states of the system under consideration, whereas the set of final states are the configurations representing, so to say, the *minimality violations* of the property of mutual exclusion, i.e., the *unsafe* states. □

As the notion of rewriting in transition systems is symmetrical, in principle it would be possible to reverse the role of initial and final states in Definition 6.7. However, as the previous example suggests, the formulation of the covering problem given in Definition 6.7 is quite natural. In fact, if we think about final states as the set of *unsafe* states of a given system, Example 6.8 suggests that any state covering an unsafe one is still unsafe. This is not true, generally speaking, for the set of initial states of a system. For instance, the configuration $\mathcal{M}_I + \{cs1, cs1\}$ in Example 6.8 *is not* a legal initial state, even though it covers $\mathcal{M}_I$, which is indeed a legal initial state. In other words, we can say that the set of final (unsafe) states is *upward-closed* (meaning that s unsafe and $s \leqslant s'$ implies s' unsafe) while the set of initial states is *not upward-closed*.

We can now give a justification of our choice of a *backward* search strategy with respect to a *forward* one. In fact, according to Proposition 6.4, using a backward search strategy amounts to an iterative computation of the set of reachable states, starting from the set of *final* states. The idea is to exploit the *upward-closure* property of the set of *unsafe* states, find a suitable *symbolic* representation and use them as the set of final states of Definition 6.7. For instance, Example 6.8 suggests that multisets of the form $\{cs1, cs1\}$ can be used to *symbolically* represent the *infinite* set of states given by $\{cs1, cs1\} + \mathcal{M}$. The resulting computation is completely independent of the set of initial states. We remark that the property of upward-closure is also crucial for reasoning about *parametric* systems, i.e., systems in which the number of components (e.g. the number K of processes in Example 6.8) is not fixed *a priori*. Verification of parametric systems is one of the goals of the present thesis work.

In the following we will try to make the above intuitions more formal. We first give the following definitions.

Definition 6.9 (Upward-closure) *Let $\leqslant$ be a quasi-order on S. Then a set $W \subseteq S$ is said to be upward-closed if $w \in W$ and $w \leqslant w'$ implies $w' \in W$. Given $w \in S$, we define the upward-closure of w, written $w^{\Uparrow}$, as the set $\{s \in S \mid w \leqslant s\}$. Similarly, for a subset $W \in S$ we define $W^{\Uparrow} = \{s \in S \mid w \leqslant s,\ w \in W\}$. An upward closed set is also called an* **ideal**, *and $W^{\Uparrow}$ is called the ideal generated by W.*

From the previous definition, it follows that a set $W \subseteq S$ is upward-closed if and only if $W = W^{\Uparrow}$.

Definition 6.10 (Basis) *Let $\leqslant$ be a quasi-order on S, and $W \subseteq S$. A basis for W is a set $\overline{W}$ such that $W = \overline{W}^{\Uparrow}$.*

The notions of *upward-closed* set and *basis* are the main ingredients of the symbolic version of the verification procedure for reachability we aim at. In particular, we are interested in *finite bases*. A finite basis $\overline{W}$ can in fact be used as a symbolic representation for the (generally infinite) set of states $\overline{W}^{\Uparrow}$. Now, let us make the assumption that the set of states from which the computation begins (i.e., the set of final states F of Definition 6.7) is upward-closed, and that we can find a finite basis for it. In order for the symbolic algorithm to work, we need to ensure that the sets computed at every iteration of the algorithm in Figure 6.1 are still upward-closed, and therefore they are in turn representable by means of finite bases. For this purpose, we introduce the following notion of *compatibility* between the transition relation and the quasi-order relation of a structured transition system.

Definition 6.11 (Upward Compatibility) *Let $T = \langle S, R, \leqslant \rangle$ be a structured transition system. We say that $\leqslant$ is (upward) compatible with $\longrightarrow$ if for all $s_1, s_2, t_1 \in S$, if $s_1 \leqslant t_1$ and $s_1 \longrightarrow s_2$, there exists $t_2 \in S$ such that $t_1 \longrightarrow t_2$ and $s_2 \leqslant t_2$.*

The above notion is usually called *strict upward compatibility*. The definition could be weakened by requiring that $t_1 \longrightarrow^* t_2$ instead of $t_1 \longrightarrow t_2$, as done in [FS01]. The following result (see [FS01] for the proof) holds.

Proposition 6.12 *Let $T = \langle S, R, \leqslant \rangle$ be a structured transition system with upward compatibility, and $W \subseteq S$. If W is upward-closed, then $Pre(W)$ is upward-closed.*

We need one more assumption to ensure effectiveness of the Pre^* computation, namely the *effective pred-basis* property.

Definition 6.13 (Effective Pred-Basis) *Let $T = \langle S, R, \leqslant \rangle$ be a structured transition system with upward compatibility. We say that T has an effective pred-basis if there exists an algorithm accepting any state $s \in S$ and computing a finite basis of $Pre(s^{\Uparrow})$, denoted $pb(s)$. We extend this notation to a set of states $W \subseteq S$ as follows: $pb(W) = \bigcup_{w \in W} pb(w)$.*

We have the following proposition, whose proof can be found in [FS01].

Proposition 6.14 ([FS01]) *Let $T = \langle S, R, \leqslant \rangle$ be a structured transition system with upward compatibility and the effective pred-basis property, and $F \subseteq S$. Given a finite basis F^b for F, define a sequence $K_0, K_1, \ldots$ of sets as follows: $K_0 = F^b$ and $K_{n+1} = K_n \cup pb(K_n)$ for $n \in \mathbb{N}$. Then $(\bigcup_{i=0}^{\infty} K_i)^{\Uparrow} = Pre^*(F)$.*

Procedure Covering
input $T = (S, R, \leqslant)$: a transition system
I: a set of initial states
F: a finite basis for the set of final states
output $Pre^*(F) \cap I = \emptyset$
begin
$W := F;\ V := \emptyset$
while $W \neq \emptyset$ **do**
choose $w \in W;\ W := W \setminus \{w\}$
if $w^{\Uparrow} \cap I \neq \emptyset$ **then return** reachable
else if $\nexists w' \in V : w^{\Uparrow} \subseteq w'^{\Uparrow}$ **then**
$V := V \cup \{w\}$
$W := W \cup pb(w)$
endif
endwhile
return unreachable
end

Figure 6.3: A verification procedure for covering

We are now ready to describe a refined version of the reachability algorithm for the covering problem, based on Proposition 6.14, which is shown in Figure 6.3. Under the hypotheses of *upward compatibility* and *effective pred-basis* property, this algorithm is effective, in the sense that any iteration of the *while* loop can be computed effectively. For the sake of precision, we need some assumptions more. Namely, we assume that $\leqslant$ *is decidable*, the test $w^{\Uparrow} \cap I \neq \emptyset$ and the test $w^{\Uparrow} \subseteq w'^{\Uparrow}$ (*insertion test*) are computable for any states w, w'. Termination in general is not guaranteed (the algorithm could loop forever). In Section 6.2 we will discuss sufficient conditions ensuring termination.

6.1.2 Symbolic Verification

In Section 6.1.1 we have discussed the theoretical foundation of our approach for verification of parameterized systems. Its effectiveness relies on structured transition systems (i.e., endowed with a notion of upward-closed set), and on the ability to compute finite bases of sets of predecessors of any upward-closed set. We will see some instances of this general scheme in chapters 7 through 10. The approach taken there slightly differs from the general scheme presented in this chapter. We summarize below some points which should help the reader to better understand the connection.

- A verification problem, formalized as a covering problem for a given structured tran-

sition system $T = \langle S, R, \leqslant \rangle$ (with upward compatibility), is given;

- verification is performed by choosing a suitable *symbolic* notation for (upward-closed) infinite sets of states (a *constraint system* in the terminology of [AJ01b]). The symbolic notation exploits the upward-closure property of the sets to represent, but, as a difference with Section 6.1.1, symbolic elements need not be elements of S. The connection is provided by a *denotation* operator, usually noted $[\![\cdot]\!]$, which maps a set of symbolic elements to the corresponding (upward-closed) set in S;

- the finite pred-basis operator (denoted $pb(\cdot)$ in Figure 6.3) is implemented through a symbolic version of the predecessor operator, let it be **Pre**, which takes as input the symbolic representation of a set V and computes the symbolic representation of the set $Pre(V)$. The following property must hold for the symbolic operator to be correct: $[\![\mathbf{Pre}(W)]\!] = Pre([\![W]\!])$;

- the *insertion test* of Figure 6.3 now becomes $[\![w]\!] \subseteq [\![w']\!]$. It is implemented through a *subsumption operator*, usually noted $\sqsubseteq$, such that $w \sqsubseteq w'$ implies $[\![w]\!] \subseteq [\![w']\!]$ [1]. The operator is typically extended to *sets* of symbolic elements as follows (i.e., *pointwise*): $W \sqsubseteq W'$ iff for every $w \in W$ there exists $w' \in W'$ s.t. $w \sqsubseteq w'$.

We conclude this section by discussing two different techniques that can sometimes be useful either to enforce termination or to accelerate convergence of the verification procedure of Figure 6.3. In Appendix A we discuss further how these techniques can be integrated into an automatic verification tool.

6.1.2.1 Invariant Strengthening

Suppose we have to solve a reachability problem, I and F being the sets of initial and final states, respectively, of a given transition system, and let us call the problem *unsatisfiable* in case it is not possible to find an execution trace from a state in I to a state in F. Now, the idea of *invariant strengthening* is to transform the original problem by considering a *larger* set of final states F'. This technique is perfectly sound, as stated in the following proposition.

Proposition 6.15 *Let $T = \langle S, R \rangle$ be a transition system and $I, F, F' \subseteq S$. If the reachability problem for I and F' is unsatisfiable and $F \subseteq F'$, then the reachability problem for I and F is unsatisfiable.*

Proof Immediate by definition. □

[1] Note that our $\sqsubseteq$ operator corresponds to the $\sqsupseteq$ operator of [AJ01b]. Here and in the following chapters we prefer to follow the classical logic programming convention

In general, the reverse implication in the above theorem does not hold, i.e., finding an execution trace from I to F' does not guarantee that an execution trace from I to F exists. Using a parallel with temporal logic specifications, a property $AG\varphi$ can be transformed, by invariant strengthening, into a property $AG(\varphi \wedge \psi)$. Intuitively, the formula ψ should represent an *invariant* which is known to hold in all reachable states. Otherwise said, the expression $\neg EF\neg\varphi$ can be transformed into $\neg EF(\neg\varphi \vee \neg\psi)$, i.e., we can enlarge the set of final (unsafe) states with a formula $\neg\psi$ which *does not* hold in any reachable state. We can refine the above proposition as follows.

Proposition 6.16 *Let $T = \langle S, R\rangle$ be a transition system and $I, F, F' \subseteq S$. Suppose $F \subseteq F'$ and $Post^*(I) \cap (F' \setminus F) = \emptyset$. Then the reachability problem for I and F' is unsatisfiable if and only if the reachability problem for I and F is unsatisfiable.*

Proof By definitions. □

We conclude this section by illustrating another technique which is similar to invariant strengthening. The idea is to *dynamically enlarge* the search space of the algorithm for backward verification. Suppose we have an *abstraction* α mapping sets of states into sets of states, and such that $W \subseteq \alpha(W)$ for every set of states W. Then, at every iteration we can substitute $\alpha(Pre(W))$ for $Pre(W)$. Intuitively, this technique is perfectly sound for the same reasons invariant strengthening is. We will use this strategy in Chapter 9 for validating specifications written in LO enriched with constraints. In that case, the abstraction α works by *relaxing* constraints. Let us define the following modified predecessor operator.

Definition 6.17 *Let $T = \langle S, R\rangle$ be a transition system, and $I, F \subseteq S$. Consider the reachability problem for I and F, and a function $\alpha : \mathcal{P}(S) \to \mathcal{P}(S)$. Then we define the operator Pre_α as follows: $Pre_\alpha(W) = \alpha(Pre(W))$ for every $W \subseteq S$.*

Correctness of the abstraction technique is stated in the following proposition.

Proposition 6.18 *Let $T = \langle S, R\rangle$ be a transition system, and $I, F \subseteq S$. Consider the reachability problem for I and F, and a function $\alpha : \mathcal{P}(S) \to \mathcal{P}(S)$ such that $W \subseteq \alpha(W)$ for every $W \subseteq S$. If $Pre^*_\alpha(F) \cap I = \emptyset$, then $Pre^*(F) \cap I = \emptyset$.*

Proof By definitions. □

6.1.2.2 Pruning

The technique of *pruning* is, so to say, complementary with respect to the methodology of dynamic abstraction illustrated above. The idea is to *dynamically cut* the search space of the algorithm for backward verification. Suppose we have computed an *over-approximation* H of the reachability set. At every iteration, we can intersect the set of reachable states computed so far with the set H. Intuitively, states not in H are not reachable from the set of initial states, therefore computing their set of predecessors is completely useless. Consequently, states not in H can always be discarded. Let us define the following modified predecessor operator.

Definition 6.19 *Let $T = \langle S, R \rangle$ be a transition system, and $I, F, H \subseteq S$. Consider the reachability problem for I and F, and suppose $Post^*(I) \subseteq H$. Then we define the operator Pre_H as follows: $Pre_H(W) = Pre(W) \cap H$ for every $W \subseteq S$.*

Correctness of the pruning technique is stated in the following proposition.

Proposition 6.20 *Let $T = \langle S, R \rangle$ be a transition system, and $I, F, H \subseteq S$. Consider the reachability problem for I and F, and suppose $Post^*(I) \subseteq H$. Then $Pre^*(F) \cap I = \emptyset$ if and only if $Pre^*_H(F) \cap I = \emptyset$.*

Proof By definitions. □

6.2 Ensuring Termination

In this section we discuss some sufficient conditions which ensure termination of the verification procedure of Figure 6.3. In particular, the theoretical foundation is provided by the theory of *well-* and *better-quasi-orderings* [Mil85], presented in Section 6.2.1, which gives rise to the notion of *well-structured transition system* [FS01], presented in Section 6.2.2.

6.2.1 The Theory of Well-Quasi-Orderings

In the following we summarize some basic definitions and results on the theory of *well-* and *better-quasi-orderings* (see for instance [Hig52, Mil85, AN00]).

We remind that a *quasi-order* $\leqslant$ on a set A is a binary relation over A which is reflexive and transitive. In the following it will be denoted $(A, \leqslant)$.

Definition 6.21 (Well-Quasi-Ordering) *A quasi-order $(A, \leqslant)$ is a well-quasi-ordering (wqo) if for each infinite sequence $a_0\ a_1\ a_2\ \ldots$ of elements in A there exist indices $i < j$ such that $a_i \leqslant a_j$.*

Examples of wqos are the identity relation on a finite set or the $\leqslant$ relation on natural numbers (but not on integers).

The following propositions state the key results involving wqos and upward-closed sets.

Proposition 6.22 ([Hig52]) *Let $(A, \leqslant)$ be a wqo, and $X \subseteq A$. If X is upward-closed, then it has a finite basis.*

Proposition 6.23 *Let $(A, \leqslant)$ be a wqo. Then every infinite sequence $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ of upward-closed sets, with $I_i \subseteq A$ for every $i \in \mathbb{N}$, eventually stabilizes, i.e., there exists $k \in \mathbb{N}$ such that $I_k = I_{k+1} = I_{k+2} = \ldots$.*

The notion of **better-quasi-ordering** (bqo) implies that of wqo, i.e., any bqo is a wqo. For the sake of simplicity, we will not present the formal definition of bqo, which can be found in [Mil85]. According to the following proposition, an entire hierarchy of bqos (and therefore of wqos) can be built starting from known ones.

In what follows, let $\widehat{n}$ denote the set $\{1, \ldots, n\}$, for $n \in \mathbb{N}$. A multiset over A can be represented as a mapping $w : \widehat{|w|} \to A$, where $|w|$ is its cardinality. Similarly, a string over A can be represented as a mapping $w : \widehat{|w|} \to A$, where $|w|$ is its length.

Proposition 6.24 (Properties of Better-Quasi-Orderings)

i. Each bqo is a wqo;

ii. If A is a finite set, then $(A, =)$ is a bqo;

iii. let $(A, \leqslant)$ be a bqo, and let $\mathcal{MS}(A)$ denote the set of finite multisets over A. Then, $(\mathcal{MS}(A), \leqslant^B)$ is a bqo, where $\leqslant^B$ is the quasi-order on $\mathcal{MS}(A)$ defined as follows: $w \leqslant^B w'$ iff there exists an injection $h : \widehat{|w|} \to \widehat{|w'|}$ such that $w(j) \leqslant w'(h(j))$ for $1 \leqslant j \leqslant |w|$;

iv. let $(A, \leqslant)$ be a bqo, and let A^* denote the set of finite strings over A. Then, $(A^*, \leqslant^*)$ is a bqo, where $\leqslant^*$ is the quasi-order on A^* defined in the following way: $w \leqslant^* w'$ iff there exists a strictly monotone (meaning that $j_1 < j_2$ iff $h(j_1) < h(j_2)$) injection $h : \widehat{|w|} \to \widehat{|w'|}$ such that $w(j) \leqslant w'(h(j))$ for $1 \leqslant j \leqslant |w|$;

v. let $(A, \leqslant)$ be a bqo. Then, $(\mathcal{P}(A), \sqsupseteq)$ is a bqo, where $\mathcal{P}(A)$ is the power set of A and $\sqsupseteq$ is the pointwise *ordering defined as follows: given $X, Y \subseteq A$, $X \sqsupseteq Y$ iff for every $y \in Y$ there exists $x \in X$ s.t. $x \leqslant y$* [2].

The reader can verify that the ordering $\leqslant^B$ of property *iii* coincides with the sub-multiset relation $\preccurlyeq$ (see Definition 2.1) when $\leqslant$ is interpreted as equality on a finite set A, so that for instance $(\mathcal{MS}(A), \preccurlyeq)$ is a wqo. This result goes under the name of Dickson's Lemma:

Proposition 6.25 (Dickson's Lemma [Dic13]) *Let $\mathcal{A}_1, \mathcal{A}_2, \ldots$ be an infinite sequence of multisets over the finite set A. Then there exist two indices i and j such that $i < j$ and $\mathcal{A}_i \preccurlyeq \mathcal{A}_j$.*

We note that properties *ii*, *iii*, and *iv* of Proposition 6.24 also hold for wqos. On the contrary, property *v* is the distinguished feature of bqos w.r.t. wqos, i.e., in general $(A, \leqslant)$ being a wqo does not imply that $(\mathcal{P}(A), \sqsupseteq)$ is a wqo (see Rado's counterexample in [AN00]). The proof of property *v* can be found in [Mar99].

Remark 6.26 Consider a wqo $(A, \leqslant)$ and the following ordering $\sqsupseteq^s$ on $\mathcal{P}(A)$: given $X, Y \subseteq A$, $X \sqsupseteq^s Y$ iff for every $x \in X$ there exists $y \in Y$ s.t. $x \leqslant y$. Then $(\mathcal{P}(A), \sqsupseteq^s)$ is a wqo. This conclusion follows by property *iii* of Proposition 6.24 (which can be reformulated for *sets* instead of *multisets*) and by observing that the existence of an injection between $X, Y \subseteq A$, satisfying the hypotheses of property *iii*, implies $X \sqsupseteq^s Y$. We remark, however, that $(\mathcal{P}(A), \sqsupseteq^s)$ being a wqo does not imply that $(\mathcal{P}(A), \sqsupseteq)$ (defined in property *v* of Proposition 6.24) is a wqo: the ordering relations $\sqsupseteq^s$ and $\sqsupseteq$ are not comparable, as the reader can verify.

6.2.2 Well-Structured Transition Systems

Building on the concept of well-quasi-ordering of Section 6.2.1, we can now extend the definition of structured transition system as follows.

Definition 6.27 (Well-Structured Transition System [FS01]) *A well-structured transition system is a structured transition system $T = \langle S, R, \leqslant \rangle$ with upward compatibility, and such that $(S, \leqslant)$ is a well-quasi-ordering.*

We have the following result.

[2] See note on page 74

reachability problems	**linear logic semantics**
transition system	proof system
state	sequent
transition	rule instance
final state { single / upward-closed	axiom { with $\mathbf{1}$ / with $\top$
reachability	provability
Pre operator	T_P operator
Pre^* operator	$lfp(T_P)$

Table 6.1: A parallel between reachability problems and linear logic semantics

Proposition 6.28 ([FS01]) *Let $T = \langle S, R, \leqslant \rangle$ be a well-structured transition system with the effective pred-basis property and decidable $\leqslant$. Then it is possible to compute a finite basis of $Pre^*(F)$ for any upward-closed set $F \subseteq S$ given via a finite basis. It follows that the covering problem is decidable for T.*

The proof of the previous proposition, which can be found in [FS01], is carried out by showing that the sequence $K_0, K_1, \ldots$ of Proposition 6.14 stabilizes for a given $m \in \mathbb{N}$, thanks to the well-quasi-ordering property of $\leqslant$.

Proposition 6.28 and the theory of well-quasi-orderings of Section 6.2.1 therefore provide sufficient conditions for the termination of the algorithm for the covering problem presented in Figure 6.3. We will see some applications in the following chapters. For instance, termination of the bottom-up evaluation procedure for propositional LO, presented in Chapter 7, is a consequence of Dickson's lemma 6.25.

6.3 Verification in Linear Logic

In this section we will explain, very informally, the connection between the reachability and covering problems presented in this chapter and provability in linear logic. This should help the reader to better understand the following chapters.

We present in Table 6.1 a succinct and suggestive view of this connection. The main idea underlying the connection is that a proof system, like the sequent systems presented in Chapter 3, can be seen as a transition system. In this view, reachability between configurations amounts to provability of a given goal from a given axiom. Axioms represent

final states, and the set of logical consequences of the axioms correspond to the *backward* reachability set, i.e., the set of states which are backward reachable from the set of final states. Computing the closure Pre^* of the predecessor operator, starting from the set of final states, corresponds to evaluating the *bottom-up semantics* of a linear logic program. Evaluation of the bottom-up semantics is carried out starting from the axioms and iteratively accumulating provable goals, exactly like computation of Pre^* starts from the set of final (unsafe) states and accumulates reachable states. A single state can be represented by an $\mathrm{LO}_{\mathbf{1}}$ axiom like $H \circ\!\!- \mathbf{1}$ (see Section 3.3.1), whereas the notion of upward-closed set of states has a counterpart in LO axioms of the form $H \circ\!\!- \top$ (see Section 3.2). In fact, we remind that in LO (without $\mathbf{1}$) provable goals are upward-closed, according to Proposition 3.9 (admissibility of the weakening rule). The counterpart of the Pre operator is a classical T_P-like fixpoint operator for logic programs [Llo87].

6.4 Related Work

In this section we briefly discuss some related work. As mentioned at the beginning of this chapter, part of the material presented here is freely taken from [FS01, AJ01b].

In [FS01], Finkel and Schnoebelen build upon previous works like [Fin87, ACJT96] and present a general theory for *well-structured transition systems*. They discuss a number of verification problems, among which the *covering* problem we have presented in this chapter, the *inevitability* problem and the *boundedness* problem. They divide resolution methods into *set-saturation* methods (like the one for the covering problem presented here) and *tree-saturation* methods, and they discuss related conditions for decidability and effectiveness. Finally, they show examples of well-structured transition systems from the fields of Petri nets, rewriting systems, process algebras, and finite state machines. In [AJ01b], Abdulla and Jonsson discuss strategies for solving the covering problem, which can be seen as an instance of the general scheme of [FS01]. They use the theory of well-quasi-orderings and show how to build an hierarchy of wqos. They focus on *symbolic* verification methods and show examples of different *constraint systems*.

The general framework of [FS01, AJ01b] has been applied to solving a number of problems for infinite-state systems. Among them, networks of timed processes [AJ98, AN01], broadcast protocols [EFM99, DEP99], cache coherence protocols [Del00], and protocols using communication over unbounded channels [AJ01a]. A discussion on the use of better-quasi-orderings and their advantage over well-quasi-orderings for verification of infinite-state systems can be found in [AN00].

Summary of the Chapter. *In this chapter we have discussed the theoretical*

foundation of a symbolic verification algorithm which can be considered the core of the present thesis work. The algorithm implements a backward-style model checking verification procedure for solving reachability problems in transition systems and it is particularly suitable to solve the so-called covering problem. We have explained why backward reachability is more convenient than forward reachability.

Following the general scheme presented in this chapter, in the following chapters we will discuss bottom-up semantics for different fragments of linear logic programs. In particular, in chapters 7 through 9 we will discuss bottom-up evaluation for propositional LO program, with or without the constant **1**, *first-order LO programs, and LO programs with constraints. For propositional LO programs, we will be able to prove termination using the well-quasi-ordering theory discussed in this chapter. We will exemplify the backward-style approach solving different problems taken, e.g., from concurrency theory or security protocols.*

Chapter 7

Bottom-Up Evaluation of Propositional LO Programs

The proof-theoretical semantics of LO we have seen in Chapter 3 corresponds to the *top-down* operational semantics based on resolution for traditional logic programming languages like Prolog. We are now interested in finding a suitable definition of *bottom-up* semantics that can be used as an alternative operational semantics for LO. More precisely, given an LO program P we would like to define a procedure to compute all goal formulas G such that G is provable from P. This procedure should enjoy the usual properties of classical bottom-up semantics, in particular its definition should be based on an *effective* fixpoint operator (i.e., at least every single step must be finitely computable), and it should be *goal-independent*. As usual in Logic Programming, goal independence is achieved by searching for proofs starting from the axioms (the unit clauses of Section 3.2) and accumulating goals which can be proved by applying program clauses to the current interpretation.

In this chapter we fist deal with a propositional formulation of LO comprising the logical connectives $\parr$, $\multimap$, $\&$ and $\top$. As proved in Section 4.3, there is a natural translation of Petri nets into this logical fragment. In particular, a Petri net transition corresponds to an LO clause, and *firing* of a transition corresponds to *backchaining* over a given clause. In this view, the bottom-up semantics for propositional LO which we will discuss in this chapter gives us an algorithmic procedure to compute the (backward) *reachability set* of a given Petri net (it can be seen as an alternative to the Karp and Miller's *coverability graph* construction [KM69]). Now, as explained in Chapter 6 (see in particular Section 6.3), the connection with the problem of *verification* is as follows. If we look at the set of axioms of a given LO program P as specifying the set of *unsafe* (final) configurations of a given system (e.g. markings in the case of Petri nets), then computing the bottom-up semantics of an LO program amounts to computing the (backward) reachability set of the system. In this view, the *immediate consequence operator* (see Section 2.5) for LO programs

which we will define in this chapter, is the counterpart of the *predecessor operator* for the given transition system. Thanks to Proposition 3.9, LO axioms can be seen as specifying *upward-closed* sets of configurations, therefore evaluating the bottom-up semantics for an LO program gives us an algorithm to solve the so-called *covering problem* (see Definition 6.7). Proposition 3.9 is crucial to prove that the algorithm for bottom-up evaluation of LO programs is terminating. This result will follow from an argument (Dickson's Lemma) which is based on the general theory of well-quasi-orderings discussed in Section 6.2.1. We will prove soundness and completeness results of the resulting semantics from scratch, without relying on Chapter 6. We will exemplify the connection between provability and verification, informally explained above, with the examples of Section 7.3.

Technically, the definition of the bottom-up semantics for LO will proceed as follows. The semantics is based on the definition of an extended notion of interpretation comprising multisets of atoms, and of a fixpoint operator which turns out to be monotonic and continuous over the complete lattice corresponding to the interpretation domain. Following the semantic framework of (constraint) logic programming [GDL95, JM94], we formulate the bottom-up evaluation procedure in two steps. More precisely, we first present a simple, non-effective notion of (concrete) interpretations and the corresponding definition of fixpoint operator, which we call T_P. Thanks to Proposition 3.9, already in the propositional case there are infinitely many provable multisets of atomic formulas. To circumvent this problem, we present an extended notion of (abstract) interpretations, ordered according to the multiset inclusion relation of their elements, and we define a symbolic and effective version of the fixpoint operator, which we call S_P. The purpose of this double definition is to ease the proof of soundness and completeness. Namely, this latter proof is carried out by proving that the fixpoint of the T_P operator is equivalent to the operational semantics and that S_P is a symbolic version of T_P. Dickson's Lemma [Dic13] ensures the termination of the fixpoint computation based on S_P for propositional LO programs.

This chapter is based on [BDM00, BDM02].

7.1 A Bottom-Up Semantics for LO

Some notations. In the following we will extensively use operations on multisets. We assume as given a fixed signature Σ comprising a *finite* set of propositional symbols. Multisets over Σ will be hereafter called *facts*, and noted $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... A multiset with (possibly duplicated) elements $a_1, \ldots, a_n \in \Sigma$ will be simply indicated as $\{a_1, \ldots, a_n\}$, overloading the usual notation for sets. We will use $\Delta, \Theta, \ldots$ to denote *contexts*, i.e., multisets of (possibly compound) goal formulas (a *fact* is a context in which every formula is atomic). Given two multisets Δ and Θ, $\Delta \preccurlyeq \Theta$ indicates multiset inclusion, $\Delta + \Theta$ (or Δ, Θ if there is no ambiguity) multiset union, and $\Delta, \{G\}$ is written simply Δ, G. Finally, we remind that,

$$\frac{}{P \vdash \top, \Delta}\ \top_r \qquad \frac{P \vdash G_1, G_2, \Delta}{P \vdash G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash G_1, \Delta \quad P \vdash G_2, \Delta}{P \vdash G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash \Delta}{P \vdash \bot, \Delta}\ \bot_r \qquad \frac{P \vdash G, \mathcal{A}}{P \vdash \widehat{H}, \mathcal{A}}\ bc\ \ (H \circ\!\!- G\ \in P)$$

Figure 7.1: A proof system for propositional LO

given a linear disjunction of atomic formulas $H = a_1 \parr \ldots \parr a_n$, the notation $\widehat{H}$ stands for the multiset $a_1, \ldots, a_n$ (by convention, $\widehat{\bot} = \epsilon$).

Before proceeding with the definition of the bottom-up semantics, we define formally the operational *top-down* semantics of LO programs. For the convenience of the reader, a proof system for propositional LO, corresponding to the proof system presented in Figure 3.2, is recalled in Figure 7.1.

Definition 7.1 (Operational Semantics) *Given a propositional LO program P, its operational semantics, denoted $O(P)$, is given by*

$$O(P) = \{\mathcal{A} \mid \mathcal{A} \text{ is a fact and } P \vdash \mathcal{A}\}.$$

Note that, according to [And92], the information on provable *facts* from a given program P is all we need to decide whether a general goal (with possible nesting of connectives) is provable from P or not. In fact, provability of a compound goal can always be reduced to provability of a finite set of atomic multisets.

We will now discuss the *bottom-up* semantics. We give the following definitions.

Definition 7.2 (Herbrand Base B_P) *Given a propositional LO program P defined over Σ, the Herbrand base of P, denoted B_P, is given by*

$$B_P = \mathcal{MS}(\Sigma) = \{\mathcal{A} \mid \mathcal{A} \text{ is a multiset (fact) over } \Sigma\}.$$

Definition 7.3 (Herbrand Interpretations) *We say that $I \subseteq B_P$ is a Herbrand interpretation. Herbrand interpretations form a complete lattice $\langle \mathcal{D}, \subseteq \rangle$ with respect to set inclusion, where $\mathcal{D} = \mathcal{P}(B_P)$.*

Before introducing the formal definition of the *ground* bottom-up semantics, we need to define a notion of satisfiability of a context Δ in a given interpretation I. For this purpose, we introduce the judgment $I \models \Delta \blacktriangleright \mathcal{A}$, where I is an interpretation, Δ is a context and $\mathcal{A}$ is

an output fact. The need for this judgment, with respect to the familiar logic programming setting [GDL95], is motivated by the arbitrary nesting of connectives in LO clause bodies, which is not allowed in traditional presentations of (constraint) logic programs. In $I \models \Delta \blacktriangleright \mathcal{A}$, $\mathcal{A}$ should be read as an *output* fact such that $\mathcal{A} + \Delta$ is valid in I. This notion of *satisfiability* is modeled according to the right-introduction rules of the connectives. In other words, the computation performed by the satisfiability judgment corresponds to *top-down* steps inside our *bottom-up* semantics. The notion of output fact $\mathcal{A}$ will simplify the presentation of the algorithmic version of the judgment which we will present in Section 7.2.

Definition 7.4 (Satisfiability Judgment) *Let P be a propositional LO program and I an interpretation. The satisfiability judgment $\models$ is defined as follows:*

$$
\begin{array}{l}
I \models \top, \Delta \blacktriangleright \mathcal{A}' \text{ for any fact } \mathcal{A}';\\
I \models \mathcal{A} \blacktriangleright \mathcal{A}' \text{ if } \mathcal{A} + \mathcal{A}' \in I;\\
I \models G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A} \text{ if } I \models G_1, \Delta \blacktriangleright \mathcal{A} \text{ and } I \models G_2, \Delta \blacktriangleright \mathcal{A};\\
I \models G_1 \parr G_2, \Delta \blacktriangleright \mathcal{A} \text{ if } I \models G_1, G_2, \Delta \blacktriangleright \mathcal{A};\\
I \models \bot, \Delta \blacktriangleright \mathcal{A} \text{ if } I \models \Delta \blacktriangleright \mathcal{A}.
\end{array}
$$

The satisfiability judgment $\models$ satisfies the following properties.

Lemma 7.5 *For every interpretation I, context Δ and fact $\mathcal{A}$,*

$$I \models \Delta \blacktriangleright \mathcal{A} \text{ iff } I \models \Delta, \mathcal{A} \blacktriangleright \epsilon.$$

Proof By simple induction on $\models$ definition. □

Lemma 7.6 *For any interpretations $I_1, I_2, \ldots$, context Δ, and fact $\mathcal{A}$,*

i. if $I_1 \subseteq I_2$ and $I_1 \models \Delta \blacktriangleright \mathcal{A}$ then $I_2 \models \Delta \blacktriangleright \mathcal{A}$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^{\infty} I_i \models \Delta \blacktriangleright \mathcal{A}$ then there exists $k \in \mathbb{N}$ s.t. $I_k \models \Delta \blacktriangleright \mathcal{A}$.

Proof The proof of *i* is by simple induction on $\models$ definition. The proof of *ii* is by induction on the derivation of $\bigcup_{i=1}^{\infty} I_i \models \Delta \blacktriangleright \mathcal{A}$.

- If $\Delta = \top, \Delta'$, obvious;

- if Δ is a fact, then $\bigcup_{i=1}^{\infty} I_i \models \Delta \blacktriangleright \mathcal{A}$ means $\Delta + \mathcal{A} \in \bigcup_{i=1}^{\infty} I_i$, which in turn implies that there exists k such that $\Delta + \mathcal{A} \in I_k$, therefore $I_k \models \Delta \blacktriangleright \mathcal{A}$;

- if $\Delta = G_1 \,\&\, G_2, \Delta'$, then by inductive hypothesis there exist k_1 and k_2 s.t. $I_{k_1} \models G_1, \Delta' \blacktriangleright \mathcal{A}$ and $I_{k_2} \models G_2, \Delta' \blacktriangleright \mathcal{A}$. By taking $k = max\{k_1, k_2\}$, by i we have that $I_k \models G_1, \Delta' \blacktriangleright \mathcal{A}$ and $I_k \models G_2, \Delta' \blacktriangleright \mathcal{A}$, therefore $I_k \models G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{A}$, i.e., $I_k \models \Delta \blacktriangleright \mathcal{A}$ as required;

- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

We now come to the definition of the fixpoint operator T_P.

Definition 7.7 (Fixpoint Operator T_P) *Given a propositional LO program P and an interpretation I, the fixpoint operator T_P is defined as follows:*

$$T_P(I) = \{\widehat{H} + \mathcal{A} \mid H \circ\!\!- G \in P,\ I \models G \blacktriangleright \mathcal{A}\}.$$

The following property holds.

Proposition 7.8 (Monotonicity and Continuity) *For every propositional LO program P, the fixpoint operator T_P is monotonic and continuous over the lattice $\langle \mathcal{D}, \subseteq \rangle$.*

Proof *Monotonicity.* Immediate from T_P definition and Lemma 7.6 *i*.
Continuity. We prove that T_P is finitary. Namely, given a chain of interpretations $I_1 \subseteq I_2 \subseteq \ldots$, T_P is finitary if $T_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$. Let $\mathcal{A} \in T_P(\bigcup_{i=1}^{\infty} I_i)$. By T_P definition, there exist a clause $H \circ\!\!- G \in P$ and a fact $\mathcal{B}$ s.t. $\mathcal{A} = \widehat{H} + \mathcal{B}$ and $\bigcup_{i=1}^{\infty} I_i \models G \blacktriangleright \mathcal{B}$. By Lemma 7.6 *ii*, we have that there exists $k \in \mathbb{N}$ s.t. $I_k \models G \blacktriangleright \mathcal{B}$. Again by T_P definition, we get $\mathcal{A} = \widehat{H} + \mathcal{B} \in T_P(I_k) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$. □

Monotonicity and continuity of the T_P operator imply, by Tarski's Theorem, that $lfp(T_P) = T_P{\uparrow_\omega}$. By analogy with fixpoint semantics for Horn clauses [Llo87], the fixpoint semantics of a propositional LO program P is then defined as follows.

Definition 7.9 (Fixpoint Semantics) *Given a propositional LO program P, its fixpoint semantics, denoted $F(P)$, is defined as follows:*

$$F(P) = lfp(T_P) = T_P{\uparrow_\omega}\ .$$

Intuitively, $T_P(I)$ is the set of *immediate logical consequences* of the program P and of the facts in I. In fact, if we define P_I as the program $\{A \circ\!\!- \top \mid \widehat{A} \in I\}$, the definition of T_P can be viewed as the following instance of the *cut* rule of linear logic (see Section 3.1):

$$\frac{\vdash P^\perp, {P_I}^\perp : H, G^\perp \quad \vdash P^\perp, {P_I}^\perp : G, \mathcal{A}}{\vdash P^\perp, {P_I}^\perp : H, \mathcal{A}} \; cut$$

which is equivalent to

$$\frac{\vdash (P, P_I)^\perp : H \circ\!\!- G \quad \vdash (P, P_I)^\perp : G, \mathcal{A}}{\vdash (P, P_I)^\perp : H, \mathcal{A}} \; cut$$

Note that, since $H \circ\!\!- G \in P$, the sequent $\vdash (P, P_I)^\perp : H \circ\!\!- G$ is always provable in linear logic. According to this view, $F(P)$ characterizes the set of *logical consequences* of a program P.

We conclude this section by proving the following fundamental result, which states that the fixpoint semantics is sound and complete with respect to the operational semantics.

Theorem 7.10 (Soundness and Completeness) *For every propositional LO program P, $F(P) = O(P)$.*

Proof $F(P) \subseteq O(P)$. We prove that for every $k \in \mathbb{N}$ and context Δ, if $T_P{\uparrow}_k \models \Delta \blacktriangleright \epsilon$ then $P \vdash \Delta$. The proof is by lexicographic induction on (k, h), where h is the length of the derivation of $T_P{\uparrow}_k \models \Delta \blacktriangleright \epsilon$.

- If $\Delta = \top, \Delta'$, obvious;
- if Δ is a fact, then $\Delta \in T_P{\uparrow}_k$, so that $T_P{\uparrow}_k \neq \emptyset$ and $k > 0$. By definition of T_P we have that there exists a fact $\mathcal{A}$ and a clause $H \circ\!\!- G \in P$, such that $T_P{\uparrow}_{k-1} \models G \blacktriangleright \mathcal{A}$ and $\Delta = \widehat{H}, \mathcal{A}$. By Lemma 7.5 we have that $T_P{\uparrow}_{k-1} \models G, \mathcal{A} \blacktriangleright \epsilon$, and then, by inductive hypothesis, $P \vdash G, \mathcal{A}$, therefore by LO *bc* rule, $P \vdash \widehat{H}, \mathcal{A}$, i.e., $P \vdash \Delta$;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$ then by inductive hypothesis $P \vdash G_1, \Delta'$ and $P \vdash G_2, \Delta'$, therefore $P \vdash G_1 \,\&\, G_2, \Delta'$ by LO $\&_r$ rule;
- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \perp, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

$O(P) \subseteq F(P)$. We prove that for every context Δ if $P \vdash \Delta$ then there exists $k \in \mathbb{N}$ such that $T_P{\uparrow}_k \models \Delta \blacktriangleright \epsilon$. The proof is by induction on the derivation of $P \vdash \Delta$.

- If the proof ends with an application of $\top_r$, then the conclusion is immediate;

- if the proof ends with an application of the *bc* rule, then $\Delta = A_1, \ldots, A_n, \mathcal{A}$, where $A_1, \ldots, A_n$ are atomic formulas, and there exists a clause $A_1 \parr \ldots \parr A_n \circ\!\!- G \in P$. For the uniformity of LO proofs, we can suppose $\mathcal{A}$ to be a fact. By inductive hypothesis, we have that there exists k such that $T_P{\uparrow}_k \models G, \mathcal{A} \blacktriangleright \epsilon$, then, by Lemma 7.5, $T_P{\uparrow}_k \models G \blacktriangleright \mathcal{A}$, which, by definition of T_P, in turn implies that $A_1, \ldots, A_n, \mathcal{A} \in T_P(T_P{\uparrow}_k) = T_P{\uparrow}_{k+1}$, therefore $T_P{\uparrow}_{k+1} \models A_1, \ldots, A_n, \mathcal{A} \blacktriangleright \epsilon$, i.e., $T_P{\uparrow}_{k+1} \models \Delta \blacktriangleright \epsilon$;

- if the proof ends with an application of the $\&_r$ rule, then $\Delta = G_1 \,\&\, G_2, \Delta'$ and, by inductive hypothesis, there exist k_1 and k_2 such that $T_P{\uparrow}_{k_1} \models G_1, \Delta' \blacktriangleright \epsilon$ and $T_P{\uparrow}_{k_2} \models G_2, \Delta' \blacktriangleright \epsilon$. Then, if $k = max\{k_1, k_2\}$ we have, by Lemma 7.6 *i* and monotonicity of T_P, that $T_P{\uparrow}_k \models G_1, \Delta' \blacktriangleright \epsilon$ and $T_P{\uparrow}_k \models G_2, \Delta' \blacktriangleright \epsilon$, therefore $T_P{\uparrow}_k \models G_1 \,\&\, G_2, \Delta' \blacktriangleright \epsilon$, i.e., $T_P{\uparrow}_k \models \Delta \blacktriangleright \epsilon$;

- the $\parr$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

□

We note that it is also possible to define a *model-theoretic* semantics (as for classical logic programming [GDL95]) based on the notion of *least model* with respect to a given class of models and partial order relation. In our setting, the partial order relation is simply set inclusion, while models are exactly Herbrand interpretations which satisfy program clauses, i.e., I is a model of P if and only if for every clause $H \circ\!\!- G \in P$ and for every fact $\mathcal{A}$,

$$I \models G \blacktriangleright \mathcal{A} \text{ implies } I \models H \blacktriangleright \mathcal{A}.$$

It turns out that the operational, fixpoint and model-theoretic semantics are all equivalent. We omit details. Finally, we also note that these semantics can be proved equivalent to the *phase semantics* for LO given in [AP91b].

7.2 An Effective Semantics for LO

The operator T_P defined in the previous section does not enjoy one of the crucial properties we required for our bottom-up semantics, namely its definition is *not* effective. As an example, take the program P consisting of the clause $a \circ\!\!- \top$. Then, $T_P(\emptyset)$ is the set of all multisets with *at least* one occurrence of a, which is an *infinite* set. In other words, $T_P(\emptyset) = \{\mathcal{B} \mid a \preccurlyeq \mathcal{B}\}$, where $\preccurlyeq$ is the multiset inclusion relation. In order to compute *effectively* one step of T_P, we have to find a *finite* representation of potentially infinite

sets of facts (in the terminology of [ACJT96], a *constraint system*). The previous example suggests us that a provable fact A may be used to *denote* the *ideal* generated by $\mathcal{A}$ (see Definition 6.9), i.e., the subset of B_P defined as follows: $[\![\mathcal{A}]\!] = \{\mathcal{B} \mid \mathcal{A} \preccurlyeq \mathcal{B}\}$. We extend this definition to interpretations as follows.

Definition 7.11 (Denotation of an Interpretation) *Given an interpretation I, its denotation $[\![I]\!]$ is the interpretation given by*

$$[\![I]\!] = \bigcup_{\mathcal{A} \in I} [\![\mathcal{A}]\!].$$

Two interpretations I and J are said to be equivalent, written $I \simeq J$, if and only if $[\![I]\!] = [\![J]\!]$.

Based on this idea, we define an *abstract* Herbrand base where we handle every single fact $\mathcal{A}$ as a representative element for $[\![\mathcal{A}]\!]$ (note that in the semantics of Section 7.1 the denotation of a fact $\mathcal{A}$ is $\mathcal{A}$ itself).

Definition 7.12 (Abstract Interpretation Domain) *Abstract interpretations form a complete lattice $\langle \mathcal{I}, \sqsubseteq \rangle$ where*

- $\mathcal{I} = \{[I]_\simeq \mid I \text{ is an interpretation}\}$;
- $[I]_\simeq \sqsubseteq [J]_\simeq$ *iff* $[\![I]\!] \subseteq [\![J]\!]$, *i.e., iff for every* $\mathcal{B} \in I$ *there exists* $\mathcal{A} \in J$ *such that* $\mathcal{A} \preccurlyeq \mathcal{B}$;
- *the least upper bound of* $[I]_\simeq$ *and* $[J]_\simeq$, *written* $[I]_\simeq \bigsqcup [J]_\simeq$, *is* $[I \cup J]_\simeq$;
- *the bottom and top elements are* $[\emptyset]_\simeq$ *and* $[\epsilon]_\simeq$, *respectively.*

The equivalence $\simeq$ allows us to reason modulo *redundancies*. For instance, any $\mathcal{A}$ is redundant in $\{\epsilon, \mathcal{A}\}$, which, in fact, is equivalent to $\{\epsilon\}$. It is important to note that to compare two ideals we simply need to compare their generators w.r.t. the multiset inclusion relation $\preccurlyeq$. Thus, given a *finite* set of facts we can always remove all redundancies using a polynomial number of comparisons.

Notation. For the sake of simplicity, in the rest of the paper we will often identify an interpretation I with its class $[I]_\simeq$. Furthermore, note that if $\mathcal{A} \preccurlyeq \mathcal{B}$, then $[\![\mathcal{B}]\!] \subseteq [\![\mathcal{A}]\!]$. In contrast, if I and J are two interpretations and $I \sqsubseteq J$ then $[\![I]\!] \subseteq [\![J]\!]$.

The two relations $\preccurlyeq$ and $\sqsubseteq$ are *well-quasi-orderings* (see Section 6.2.1). This is a direct consequence of Proposition 6.25 (Dickson's Lemma). This property is the key point for proving termination of the fixpoint computation for propositional LO programs. In fact, it

will allow us to prove that the computation of the least fixpoint of the *symbolic* formulation of the operator T_P (working on abstract interpretations) is guaranteed to terminate on every input LO program.

By definition of $\sqsubseteq$ and Proposition 6.23, the following corollary holds.

Corollary 7.13 *There are no infinite chains of interpretations $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ such that for all $j, k \in \mathbb{N}$ with $j < k$, $I_k \not\sqsubseteq I_j$.*

Corollary 7.13 ensures that it is not possible to generate infinite sequences of interpretations such that each element is not *subsumed* (using a terminology from constraint logic programming) by one of the previous elements in the sequence. The problem now is to define a fixpoint operator over abstract interpretations that is *correct* and *complete* w.r.t. the ground semantics. If we find it, then we can use the corollary to prove that (for any program) its fixpoint can be reached after finitely many steps. For this purpose and using the multiset operations $\setminus$ (difference), $\bullet$ (least upper bound w.r.t. $\preccurlyeq$), and ϵ (empty multiset) defined in Section 2.2, we first define a new, abstract version, of the satisfiability relation $\models$. The intuition under the new judgment $I \Vdash \Delta \blacktriangleright \mathcal{A}$, where I is an (abstract) interpretation, Δ is a context, and $\mathcal{A}$ is a fact, is that $\mathcal{A}$ is the *minimal* fact (in a sense to be clarified) that should be added to Δ in order for $\mathcal{A} + \Delta$ to be satisfiable in I.

Definition 7.14 (Abstract Satisfiability Judgment) *Let P be a propositional LO program and $I \in \mathcal{P}(B_P)$. The abstract satisfiability judgment $\Vdash$ is defined as follows:*

$$
\begin{array}{l}
I \Vdash \top, \Delta \blacktriangleright \epsilon; \\
I \Vdash \mathcal{A} \blacktriangleright \mathcal{B} \setminus \mathcal{A} \text{ for } \mathcal{B} \in I; \\
I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A}_1 \bullet \mathcal{A}_2 \text{ if } I \Vdash G_1, \Delta \blacktriangleright \mathcal{A}_1,\ I \Vdash G_2, \Delta \blacktriangleright \mathcal{A}_2; \\
I \Vdash G_1 \parr G_2, \Delta \blacktriangleright \mathcal{A} \text{ if } I \Vdash G_1, G_2, \Delta \blacktriangleright \mathcal{A}; \\
I \Vdash \bot, \Delta \blacktriangleright \mathcal{A} \text{ if } I \Vdash \Delta \blacktriangleright \mathcal{A}.
\end{array}
$$

Given a finite interpretation I and a context Δ, the previous definition gives us an *algorithm* to compute all facts $\mathcal{A}$ such that $I \Vdash \Delta \blacktriangleright \mathcal{A}$ holds.

Example 7.15 Let us consider the following clause and interpretation:

$$b \multimap (d \parr e) \,\&\, f, \quad I = \{\{c, d\}, \{c, f\}\}.$$

We want to compute the facts $\mathcal{A}$ for which $I \Vdash G \blacktriangleright \mathcal{A}$, where $G = (d \parr e) \,\&\, f$ is the body of the clause. From the second rule defining the judgment $\Vdash$, we have that $I \Vdash \{d, e\} \blacktriangleright \{c\}$, because $\{c, d\} \in I$ and $\{c, d\} \setminus \{d, e\} = \{c\}$. Therefore we get $I \Vdash d \parr e \blacktriangleright \{c\}$ using the fourth

rule for $\Vdash$. Similarly, we have that $I \Vdash \{f\} \blacktriangleright \{c\}$, because $\{c, f\} \in I$ and $\{c, f\} \backslash \{f\} = \{c\}$. By applying the third rule for $\Vdash$, with $G_1 = d \parr e$, $G_2 = f$, $\mathcal{A}_1 = \{c\}$, $\mathcal{A}_2 = \{c\}$ and $\Delta = \epsilon$, we get $I \Vdash G \blacktriangleright \{c\}$, in fact $\{c\} \bullet \{c\} = \{c\}$. There are other ways to apply the rules for $\Vdash$. In fact, we can get $I \Vdash \{d, e\} \blacktriangleright \{c, f\}$, because $\{c, f\} \in I$ and $\{c, f\} \setminus \{d, e\} = \{c, f\}$. Similarly, we can get $I \Vdash \{f\} \blacktriangleright \{c, d\}$. By considering all combinations, it turns out that $I \Vdash G \blacktriangleright \mathcal{A}$, for every $\mathcal{A} \in \{\{c\}, \{c, f\}, \{c, d\}, \{c, d, f\}\}$. The information conveyed by $\{c, f\}, \{c, d\}, \{c, d, f\}$ is in some sense *redundant*, as we will see in the following (see Example 7.26). In other words, it is not always true that the output fact of the judgment $\Vdash$ is *minimal* (in the previous example only the output $\{c\}$ is minimal). Nevertheless, the important point to be stressed here is that the set of possible facts satisfying the judgment, given I and G, is *finite*. This will be sufficient to ensure effectiveness of the fixpoint operator. □

The connection between the satisfiability judgments $\models$ and $\Vdash$ is clarified by the following lemma.

Lemma 7.16 *For every $I \in \mathcal{P}(B_P)$, context Δ, and fact $\mathcal{C}$,*

i. if $I \Vdash \Delta \blacktriangleright \mathcal{A}$, then $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}'$ for all $\mathcal{A}'$ s.t. $\mathcal{A} \preccurlyeq \mathcal{A}'$;

ii. if $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}'$, then there exists $\mathcal{A}$ such that $I \Vdash \Delta \blacktriangleright \mathcal{A}$ and $\mathcal{A} \preccurlyeq \mathcal{A}'$.

Proof

i. By induction on the derivation of $I \Vdash \Delta \blacktriangleright \mathcal{A}$.

- $I \Vdash \top, \Delta \blacktriangleright \epsilon$ and $[\![I]\!] \models \top, \Delta \blacktriangleright \mathcal{A}'$ and $\epsilon \preccurlyeq \mathcal{A}'$ for any $\mathcal{A}'$;
- if $I \Vdash \mathcal{A} \blacktriangleright \mathcal{A}'$ then $\mathcal{A}' = \mathcal{B} \backslash \mathcal{A}$ for $\mathcal{B} \in I$. Since $\mathcal{B} \preccurlyeq (\mathcal{B} \backslash \mathcal{A}) + \mathcal{A} = \mathcal{A}' + \mathcal{A}$, we have that $(\mathcal{B} \backslash \mathcal{A}) + \mathcal{A} \in [\![I]\!]$, therefore $[\![I]\!] \models \mathcal{A} \blacktriangleright \mathcal{B} \backslash \mathcal{A}$, so that $[\![I]\!] \models \mathcal{A} \blacktriangleright \mathcal{C}$ for all C s.t. $\mathcal{A}' = \mathcal{B} \backslash \mathcal{A} \preccurlyeq \mathcal{C}$, because $[\![I]\!]$ is upward closed;
- if $I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A}$ then $\mathcal{A} = \mathcal{A}_1 \bullet \mathcal{A}_2$ and $I \Vdash G_1, \Delta \blacktriangleright \mathcal{A}_1$ and $I \Vdash G_2, \Delta \blacktriangleright \mathcal{A}_2$. By inductive hypothesis, $[\![I]\!] \models G_1, \Delta \blacktriangleright \mathcal{B}_1$ and $[\![I]\!] \models G_2, \Delta \blacktriangleright \mathcal{B}_2$ for any $\mathcal{B}_1, \mathcal{B}_2$ s.t. $\mathcal{A}_1 \preccurlyeq \mathcal{B}_1$ and $\mathcal{A}_2 \preccurlyeq \mathcal{B}_2$. That is, $[\![I]\!] \models G_i, \Delta \blacktriangleright \mathcal{C}$ for any $\mathcal{C} \in [\![\mathcal{A}_1 \bullet \mathcal{A}_2]\!]$ $i : 1, 2$. It follows that $[\![I]\!] \models G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{C}$ for all $\mathcal{C} \in [\![\mathcal{A}_1 \bullet \mathcal{A}_2]\!]$;
- the $\parr$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}'$.

- The $\top$-case follows by definition;

- if $[\![I]\!] \models \mathcal{A} \blacktriangleright \mathcal{A}'$ then $\mathcal{A}' + \mathcal{A} \in [\![I]\!]$, i.e., there exists $\mathcal{B} \in I$ s.t. $\mathcal{B} \preccurlyeq \mathcal{A}' + \mathcal{A}$. Since $\mathcal{B} \setminus \mathcal{A} \preccurlyeq (\mathcal{A}' + \mathcal{A}) \setminus \mathcal{A} = \mathcal{A}'$, it follows that for $\mathcal{C} = \mathcal{B} \setminus \mathcal{A}$, $I \Vdash \mathcal{A} \blacktriangleright \mathcal{C}$ and $\mathcal{C} \preccurlyeq \mathcal{A}'$;
- if $[\![I]\!] \models G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A}$ then $[\![I]\!] \models G_i, \Delta \blacktriangleright \mathcal{A}$ for $i : 1, 2$. By inductive hypothesis, there exists $\mathcal{A}_i$ such that $\mathcal{A}_i \preccurlyeq \mathcal{A}$, $I \Vdash G_i, \Delta \blacktriangleright \mathcal{A}_i$ for $i : 1, 2$, i.e., $I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A}_1 \bullet \mathcal{A}_2$. The thesis follows noting that $\mathcal{A}_1 \bullet \mathcal{A}_2 \preccurlyeq \mathcal{A}$;
- the $\parr$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

□

The satisfiability judgment $\Vdash$ also satisfies the following properties.

Lemma 7.17 *For any $I_1, I_2, \ldots \in \mathcal{P}(B_P)$, context Δ, and fact $\mathcal{A}$,*

i. if $I_1 \sqsubseteq I_2$ and $I_1 \Vdash \Delta \blacktriangleright \mathcal{A}$ then there exists $\mathcal{A}'$ such that $I_2 \Vdash \Delta \blacktriangleright \mathcal{A}'$ and $\mathcal{A}' \preccurlyeq \mathcal{A}$;

ii. if $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, and $\bigsqcup_{i=1}^{\infty} I_i \Vdash \Delta \blacktriangleright \mathcal{A}$ then there exists $k \in \mathbb{N}$ s.t $I_k \Vdash \Delta \blacktriangleright \mathcal{A}$.

Proof

i. If $I_1 \Vdash \Delta \blacktriangleright \mathcal{A}$, then by Lemma 7.16 *i*, $[\![I_1]\!] \models \Delta \blacktriangleright \mathcal{A}$. Since $[\![I_1]\!] \subseteq [\![I_2]\!]$ then, by Lemma 7.6 *i*, $[\![I_2]\!] \models \Delta \blacktriangleright \mathcal{A}$. Thus, by Lemma 7.16 *ii*, there exists $\mathcal{A}' \preccurlyeq \mathcal{A}$ s.t. $I_2 \Vdash \Delta \blacktriangleright \mathcal{A}'$;

ii. If $\bigsqcup_{i=1}^{\infty} I_i \Vdash \Delta \blacktriangleright \mathcal{A}$, then by Lemma 7.16 *i*, $[\![\bigsqcup_{i=1}^{\infty} I_i]\!] \models \Delta \blacktriangleright \mathcal{A}$, i.e., as it can be readily verified from Definition 7.11 and Definition 7.12, $\bigcup_{i=1}^{\infty} [\![I_1]\!] \models \Delta \blacktriangleright \mathcal{A}$. By Lemma 7.6 *ii*, there exists $k \in \mathbb{N}$ s.t. $[\![I_k]\!] \models \Delta \blacktriangleright \mathcal{A}$. Thus, by Lemma 7.16 *ii*, there exists $\mathcal{A}' \preccurlyeq \mathcal{A}$ s.t. $I_k \Vdash \Delta \blacktriangleright \mathcal{A}'$.

□

We are ready now to define the abstract fixpoint operator $S_P : \mathcal{I} \to \mathcal{I}$. We will proceed in two steps. We will first define an operator working over elements of $\mathcal{P}(B_P)$. With a little bit of overloading, we will call the operator with the same name, i.e., S_P. As for the S_P operator used in the symbolic semantics of CLP programs [JM94], the operator should satisfy the equation $[\![S_P(I)]\!] = T_P([\![I]\!])$ for every $I \in \mathcal{P}(B_P)$. This property ensures the soundness and completeness of the *symbolic* representation w.r.t. the concrete one.

After defining the operator over $\mathcal{P}(B_P)$, we will lift it to our abstract domain $\mathcal{I}$ consisting of the equivalence classes of elements of $\mathcal{P}(B_P)$ w.r.t. the relation $\simeq$ defined in Definition 7.11. Formally, we first introduce the following definition.

Definition 7.18 (Symbolic Fixpoint Operator S_P) *Given a propositional LO program P and $I \in \mathcal{P}(B_P)$, the symbolic fixpoint operator S_P is defined as follows:*

$$S_P(I) = \{\widehat{H} + \mathcal{A} \mid H \circ\!\!- G \in P,\ I \Vdash G \blacktriangleright \mathcal{A}\}.$$

The following property shows that S_P is sound and complete w.r.t. T_P.

Proposition 7.19 *For every propositional LO program P and $I \in \mathcal{P}(B_P)$, $[\![S_P(I)]\!] = T_P([\![I]\!])$.*

Proof Let $\mathcal{A} = \widehat{H}, \mathcal{B} \in S_P(I)$ where $H \circ\!\!- G \in P$ and $I \Vdash G \blacktriangleright \mathcal{B}$ then, by Lemma 7.16 *i*, $[\![I]\!] \models G \blacktriangleright \mathcal{B}'$ for any $\mathcal{B}'$ s.t. $\mathcal{B} \preccurlyeq \mathcal{B}'$. Thus, for any $\mathcal{A}' = \widehat{H}, \mathcal{B}'$ s.t. $\mathcal{A} \preccurlyeq \mathcal{A}'$, $\mathcal{A}' \in T_P([\![I]\!])$. Vice versa, if $\mathcal{A} \in T_P([\![I]\!])$ then $\mathcal{A} = \widehat{H}, \mathcal{B}$ where $H \circ\!\!- G \in P$ and $[\![I]\!] \models G \blacktriangleright \mathcal{B}$. By Lemma 7.16 *ii*, there exists $\mathcal{B}'$ s.t. $\mathcal{B}' \preccurlyeq \mathcal{B}$ and $I \Vdash G \blacktriangleright \mathcal{B}'$, i.e., $\mathcal{A}' = \widehat{H}, \mathcal{B}' \in S_P(I)$ and $\mathcal{A}' \preccurlyeq \mathcal{A}$. □

Furthermore, the following corollary holds.

Corollary 7.20 *For every propositional LO program P and $I, J \in \mathcal{P}(B_P)$, if $I \simeq J$ then $S_P(I) \simeq S_P(J)$.*

Proof If $I \simeq J$, i.e., $[\![I]\!] = [\![J]\!]$, then $T_P([\![I]\!]) = T_P([\![J]\!])$. Thus, by Prop. 7.19 it follows that $[\![S_P(I)]\!] = [\![S_P(J)]\!]$, i.e., $S_P(I) \simeq S_P(J)$. □

The previous corollary allows us to safely lift S_P definition from the lattice $\langle \mathcal{P}(B_P), \subseteq \rangle$ to the lattice $\langle \mathcal{I}, \sqsubseteq \rangle$. Formally, we define the abstract fixpoint operator as follows.

Definition 7.21 (Abstract Fixpoint Operator S_P) *Given a propositional LO program P and an equivalence class $[I]_\simeq$ of $\mathcal{I}$, the abstract fixpoint operator S_P is defined as follows:*

$$S_P([I]_\simeq) = [S_P(I)]_\simeq$$

where $S_P(I)$ is defined in Definition 7.18.

For the sake of simplicity, in the following we will use I to denote its class $[I]_\simeq$. The abstract fixpoint operator S_P satisfies the following property.

Proposition 7.22 (Monotonicity and Continuity) *For every propositional LO program P, the abstract fixpoint operator S_P is monotonic and continuous over the lattice $\langle \mathcal{I}, \sqsubseteq \rangle$.*

Procedure Symbolic Fixpoint
input P: an LO program
output $\mathcal{F}_{sym}(P)$
begin
  $New := \{\widehat{H} \mid H \circ\!\!- \top \in P\}$; $Old := \emptyset$
  repeat
    $Old := New$
    $New := S_P(New)$
  until $New \sqsubseteq Old$
return Old **end**

Figure 7.2: Symbolic fixpoint computation

Proof *Monotonicity.* For any $\mathcal{A} = \widehat{H}, \mathcal{B} \in S_P(I)$ there exists $H \circ\!\!- G \in P$ s.t. $I \Vdash G \blacktriangleright \mathcal{B}$. Assume now that $I \sqsubseteq J$. Then, by Lemma 7.17 i, we have that $J \Vdash G \blacktriangleright \mathcal{B}'$ for $\mathcal{B}' \preccurlyeq \mathcal{B}$. Thus, there exists $\mathcal{A}' = \widehat{H}, \mathcal{B}' \in S_P(J)$ such that $\mathcal{A}' \preccurlyeq \mathcal{A}$, i.e., $S_P(I) \sqsubseteq S_P(J)$.
Continuity. We show that S_P is finitary. Let $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ be a chain of interpretations. For any $\mathcal{A} = \widehat{H}, \mathcal{B} \in S_P(\bigsqcup_{i=1}^{\infty} I_i)$ there exists $H \circ\!\!- G \in P$ s.t. $\bigsqcup_{i=1}^{\infty} I_i \Vdash G \blacktriangleright \mathcal{B}$. By Lemma 7.17 ii, there exists $k \in \mathbb{N}$ s.t. $I_k \Vdash G \blacktriangleright \mathcal{B}'$ for $\mathcal{B}' \preccurlyeq \mathcal{B}$. Thus, $\mathcal{A}' = \widehat{H}, \mathcal{B}' \in S_P(I_k)$, i.e., $\mathcal{A}' \in \bigsqcup_{i=1}^{\infty} S_P(I_i)$ with $\mathcal{A}' \preccurlyeq \mathcal{A}$, i.e., $S_P(\bigsqcup_{i=1}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=1}^{\infty} S_P(I_i)$. □

Corollary 7.23 *For every propositional LO program P, $[\![lfp(S_P)]\!] = lfp(T_P)$.*

Let $\mathcal{F}_{sym}(P) = lfp(S_P)$, then we have the following main theorem.

Theorem 7.24 (Soundness and Completeness) *For every propositional LO program P, $O(P) = F(P) = [\![\mathcal{F}_{sym}(P)]\!]$. Furthermore, there exists $k \in \mathbb{N}$ such that $\mathcal{F}_{sym}(P) = \bigsqcup_{i=0}^{k} S_P{\uparrow_k}(\emptyset)$.*

Proof Theorem 7.10 and Corollary 7.23 show that $O(P) = F(P) = [\![\mathcal{F}_{sym}(P)]\!]$. Corollary 7.13 guarantees that the fixpoint of S_P can always be reached after finitely many steps. □

The previous results give us an algorithm to compute the operational and fixpoint semantics of a propositional LO program via the operator S_P. The algorithm is inspired by the *backward reachability* algorithm used in [ACJT96, FS01] to compute *backwards* the closure of the *predecessor* operator of a well-structured transition system. The algorithm in pseudo-code for computing $F(P)$ is shown in Figure 7.2. Corollary 7.13 guarantees that the algorithm always terminates and returns a *symbolic* representation of $O(P)$. As a corollary of Theorem 7.24, we obtain the following result.

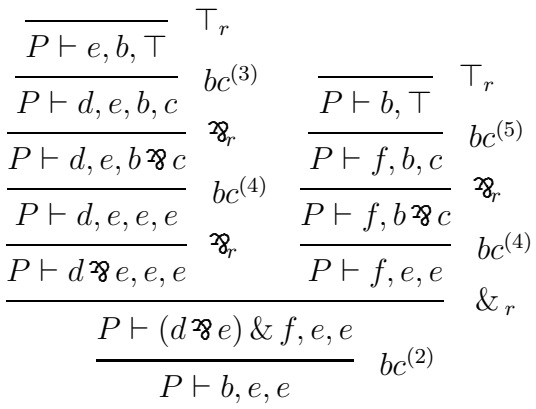

Figure 7.3: Another example of LO derivation

Corollary 7.25 *The provability of $P \vdash G$ in propositional LO is* decidable.

In view of Proposition 3.9, this result can be considered as an instance of the general decidability result [Kop95] for propositional *affine* linear logic (i.e., linear logic with *weakening*).

Example 7.26 We compute the fixpoint semantics for the program P of Example 3.10, which is given below.

1. $a \circ\!\!- b \parr c$
2. $b \circ\!\!- (d \parr e) \,\&\, f$
3. $c \parr d \circ\!\!- \top$
4. $e \parr e \circ\!\!- b \parr c$
5. $c \parr f \circ\!\!- \top$

We start the computation from $S_P{\uparrow}_0 = \emptyset$. The first step consists in adding the multisets corresponding to program facts, i.e., clauses 3 and 5, therefore we compute

$$S_P{\uparrow}_1 = \{\{c, d\}, \{c, f\}\}.$$

Now, we can try to apply clauses 1, 2, and 4 to facts in $S_P{\uparrow}_1$. From the first clause, we have that $S_P{\uparrow}_1 \Vdash \{b, c\} \blacktriangleright \{d\}$ and $S_P{\uparrow}_1 \Vdash \{b, c\} \blacktriangleright \{f\}$, therefore $\{a, d\}$ and $\{a, f\}$ are elements of $S_P{\uparrow}_2$. Similarly, for clause 2 we have that $S_P{\uparrow}_1 \Vdash \{d, e\} \blacktriangleright \{c\}$ and $S_P{\uparrow}_1 \Vdash \{f\} \blacktriangleright \{c\}$, therefore we have, from the rule for $\&$, that $\{b, c\}$ belongs to $S_P{\uparrow}_2$ (we can also derive other judgments for clause 2, as seen in Example 7.15, for instance $S_P{\uparrow}_1 \Vdash \{d, e\} \blacktriangleright \{c, f\}$, but it immediately turns out that all these judgments give rise to *redundant* information, i.e., facts that are subsumed by the already calculated ones). By clause 4, finally we have that $S_P{\uparrow}_1 \Vdash \{b, c\} \blacktriangleright \{d\}$ and $S_P{\uparrow}_1 \Vdash \{b, c\} \blacktriangleright \{f\}$, therefore $\{d, e, e\}$ and $\{e, e, f\}$ belong to $S_P{\uparrow}_2$. We can therefore take the following equivalence class as representative for $S_P{\uparrow}_2$:

$$S_P{\uparrow}_2 = \{\{c, d\}, \{c, f\}, \{a, d\}, \{a, f\}, \{b, c\}, \{d, e, e\}, \{e, e, f\}\}.$$

We can similarly calculate $S_P{\uparrow}_3$. For clause 1 we immediately have that $S_P{\uparrow}_2 \Vdash \{b, c\} \blacktriangleright \epsilon$, so that $\{a\}$ is an element of $S_P{\uparrow}_3$; this makes the information given by $\{a, d\}$ and $\{a, f\}$ in $S_P \uparrow_2$ redundant. From clause 4 we can get that $\{e, e\}$ is another element of $S_P \uparrow_3$, which implies that the information given by $\{d, e, e\}$ and $\{e, e, f\}$ is now redundant. No additional (not redundant) elements are obtained from clause 2. We therefore can take

$$S_P{\uparrow}_3 = \{\{c, d\}, \{c, f\}, \{b, c\}, \{a\}, \{e, e\}\}.$$

The reader can verify that $S_P{\uparrow}_4 = S_P{\uparrow}_3 = \mathcal{F}_{sym}(P)$ so that

$$O(P) = F(P) = [\![\{\{c, d\}, \{c, f\}, \{b, c\}, \{a\}, \{e, e\}\}]\!].$$

We suggest the reader to compare the top-down proof for the goal e, e, given in Figure 3.3, and the part of the bottom-up computation which yields the same goal. The order in which the backchaining steps are performed is reversed, as expected. Moreover, the top-down computation requires to solve one goal, namely d, e, c, which is not *minimal*, in the sense that its proper subset c, d is still provable. Using the bottom-up algorithm sketched above, at every step only the minimal information (in this case c, d) is kept at every step.

In general, this strategy has the further advantage of reducing the amount of non-determinism in the proof search. For instance, let us consider the goal b, e, e (which is certainly provable because of Proposition 3.9). This goal has at least two different proofs. The first is a slight modification of the proof in Figure 3.3 (just add the atom b to every sequent). An alternative proof, obtained by changing the order of applications of the backchaining steps, is given in Figure 7.3. There are even more complicated proofs (for instance in the left branch we could rewrite b again by backchaining over clause 2 rather than axiom 3). The bottom-up computation avoids these complications by keeping only *minimal* information at every step. We would also like to stress that the bottom-up computation is always guaranteed to terminate, as stated in Theorem 7.24, while in general the top-down computation can diverge. □

7.3 Examples

We conclude this chapter by showing how computation of the bottom-up semantics for propositional LO programs can be used to prove *safety* properties. In particular, we consider the producer/consumer example of Section 4.2.1 and the net for mutual exclusion of

$\{free,\ init\}$
$\{buffer,\ init\}$
$\{init,\ init\}$
$\{buffer,\ buffer,\ buffer,\ buffer,\ buffer,\ buffer\}$
$\{free,\ free,\ free,\ free,\ free,\ free,\ producing\}$
$\{buffer,\ free,\ free,\ free,\ free,\ free,\ producing\}$
$\{buffer,\ buffer,\ free,\ free,\ free,\ free,\ producing\}$
$\{buffer,\ buffer,\ buffer,\ free,\ free,\ free,\ producing\}$
$\{buffer,\ buffer,\ buffer,\ buffer,\ free,\ free,\ producing\}$
$\{buffer,\ buffer,\ buffer,\ buffer,\ buffer,\ free,\ producing\}$
$\{buffer,\ buffer,\ buffer,\ buffer,\ buffer,\ releasing,\ free\}$
$\{free,\ buffer,\ buffer,\ buffer,\ buffer,\ releasing,\ free\}$
$\{free,\ free,\ buffer,\ buffer,\ buffer,\ releasing,\ free\}$
$\{free,\ free,\ free,\ buffer,\ buffer,\ releasing,\ free\}$
$\{free,\ free,\ free,\ free,\ buffer,\ releasing,\ free\}$
$\{free,\ free,\ free,\ free,\ free,\ releasing,\ free\}$

Figure 7.4: Fixpoint computed for the producer/consumer example

Section 6.1.1. We use these examples to exemplify our methodology for backward verification of safety properties. Other examples will be presented in the following chapters. Verification has been performed by means of an automatic tool implementing the procedure for the covering problem described in Figure 6.3. We refer the reader to Appendix A for a description of this tool and details about the environment in which the experiments have been conducted.

7.3.1 A Producer/Consumer Example

Let us consider the Petri net of Figure 4.1, representing a producer/consumer net with a bounded capacity buffer. According to Section 4.3, translation of this net yields the following LO program:

1. $producing \circ\!\!- releasing$
2. $releasing \parr free \circ\!\!- producing \parr buffer$
3. $buffer \parr buffer \parr acquiring \circ\!\!- free \parr free \parr consuming$
4. $consuming \circ\!\!- acquiring$

If we wish to verify properties of the reachable states of the Petri net in Figure 4.1, first of all we need to give a specification of the initial states of the system. Specifically, we need

to encode the *initial marking* drawn in Figure 4.1. To this aim, we can simply introduce a new propositional symbol (place) *init* and add the following LO clause:

$$5.\ \ init : -releasing \parr buffer \parr buffer \parr buffer \parr free \parr free \parr acquiring$$

The only purpose of *init* is to load the initial configuration of the system.

We can now verify a *safety* property using the backward reachability algorithm of Section 6.1.1. For instance, the Petri net in Figure 4.1 has the following bounded capacity property: *at any time, the buffer contains at most five items*. We can formally verify that the above specification complies with the bounded capacity property using the following strategy. First of all, we can see the above problem as an instance of the *covering problem* for the Petri net in Figure 4.1. In fact, the bounded capacity property is equivalent to saying that *the buffer will never contain more than five items*. Therefore, we need to verify that starting from the initial marking we will never reach a marking containing six or more items in place *buffer*. Otherwise said, we have to verify that it is not possible to *cover* the following marking: $\{buffer, buffer, buffer, buffer, buffer, buffer\}$. According to the discussion of Section 6.3, we can encode this problem using the following LO axiom:

$$6.\ \ buffer \parr buffer \parr buffer \parr buffer \parr buffer \parr buffer \circ\!\!- \top$$

The head of the above axiom represents the *minimality violation* of the safety property under consideration. In other words, any marking with *at least* six items in place *buffer* is *unsafe*. We recall that, from a logical point of view, this is a result of Proposition 3.9.

We can now run our verification tool (see Appendix A) to *automatically* verify if the bounded capacity property holds. We simply need to evaluate the bottom-up semantics of the above LO program, including the specification of the initial and of the final (unsafe) states (i.e., we consider clauses 1 through 6), and check that the fixpoint does not contain any legal initial state. In this case, the initial state (marking) is represented by the atom *init*, therefore we simply need to verify that the marking $\{init\}$ is not contained in the fixpoint, i.e., that $\{init\}$ is not *backward* reachable. Termination of the bottom-up evaluation is guaranteed by Lemma 7.13. Running the verification tool, we get the fixpoint shown in Figure 7.4. The fixpoint contains 16 elements and is reached after 13 steps (iterations). As it does not contain *init*, the bounded capacity property is verified.

We have used the producer/consumer example mainly to illustrate our approach and discuss our methodology for verification of safety properties. The above example is not particularly meaningful for the following reason. As discussed in Section 4.2.2 (see Example 4.16), the following invariant holds in the Petri net of Figure 4.1: the total number of tokens in places *buffer* and *free* is always less or equal than five. The property we have discussed above is a simple consequence of this invariant, and therefore it can be computed statically. In the next section we present a more interesting example.

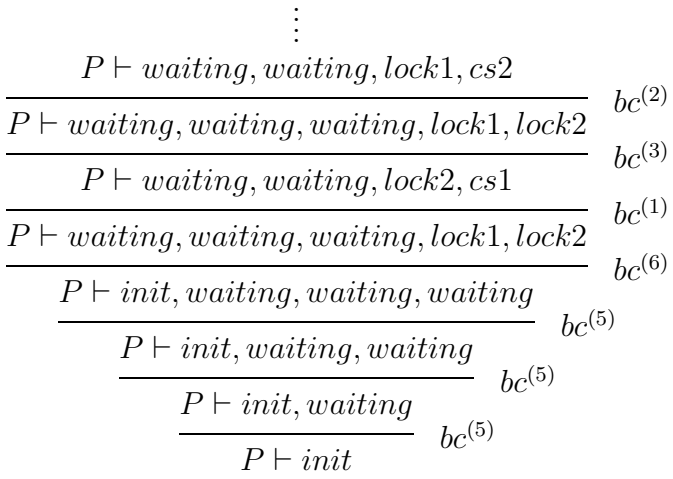

Figure 7.5: A petri net for mutual exclusion: example trace

7.3.2 A Petri Net for Mutual Exclusion

Let us consider the Petri net presented in Figure 6.2. The net represents a simple monitor for a *parameterized* system with two mutually exclusive critical sections (see discussion in Example 6.8). Two locks are used to protect the critical sections from unauthorized access. The net is parametric in the number of *waiting* processes, i.e., processes willing to enter one of two critical sections. Following the encoding of section 4.3, the above Petri net is represented by the LO program below:

1. $waiting \otimes lock1 \otimes lock2 \circ\!\!- lock2 \otimes cs1$
2. $waiting \otimes lock1 \otimes lock2 \circ\!\!- lock1 \otimes cs2$
3. $cs1 \circ\!\!- waiting \otimes lock1$
4. $cs2 \circ\!\!- waiting \otimes lock2$

The first two clauses encode transitions Enter1 and Enter2, whereas the last two clauses encode transitions Leave1 and Leave2, respectively.

Following the methodology illustrated in Section 7.3.1, we need to encode the initial and final (i.e., unsafe) states of the system. Initial states can be encoded by introducing a new propositional symbol *init* and adding the following LO clauses:

5. $init \circ\!\!- init \otimes waiting$
6. $init \circ\!\!- lock1 \otimes lock2$

The above encoding is very flexible, in that it allows us to leave the number of initial processes in place *waiting* (i.e., the *parameter* K in Figure 6.2) unspecified. In fact, by

$\{cs1,\ cs1\}$
$\{cs1,\ cs2\}$
$\{cs2,\ cs2\}$
$\{init,\ init\}$
$\{cs1,\ init\}$
$\{cs2,\ init\}$
$\{lock2,\ init\}$
$\{lock1,\ init\}$
$\{waiting,\ lock2,\ waiting,\ lock1,\ lock2\}$
$\{lock1,\ waiting,\ waiting,\ lock1,\ lock2\}$
$\{cs1,\ waiting,\ lock1,\ lock2\}$
$\{cs2,\ waiting,\ lock1,\ lock2\}$
$\{lock2,\ waiting,\ lock2,\ cs1\}$
$\{lock1,\ waiting,\ lock1,\ cs2\}$

Figure 7.6: Fixpoint computed for the mutual exclusion example

using the first clause we can put any number of processes in place *waiting*, while the second clause initializes the two locks and let the processes compete for accessing the critical sections. For clarity, an example trace (LO proof fragment) is shown in Figure 7.5 (where P is the LO program made up of clauses 1 through 6, and we have followed the usual notational conventions).

It remains to give the encoding of the unsafe states of the system. Specifically, a configuration is unsafe if *at least* two processes are accessing either of the critical sections at the same time. Namely, we have the following encoding in LO:

7. $cs1 \parr cs1 \circ\!\!- \top$
8. $cs1 \parr cs2 \circ\!\!- \top$
9. $cs2 \parr cs2 \circ\!\!- \top$

Now, we can run our automatic verification tool (see Appendix A) to evaluate the bottom-up semantics of the above LO program (clauses 1 through 9), and check whether the fixpoint contains the configuration $\{init\}$ or not. We stress that in this case static computation of structural invariants for the Petri net of Figure 6.2 is not sufficient to prove the above safety property (see for instance [DRB01]). As an aside, we remark that the matrix representation of the Petri net in Figure 6.2 (upon which the calculus of structural invariants is based) is *ambiguous*, in the sense that there are cycles (e.g. the one given by the arc from *lock*2 to Enter1 and the arc from Enter1 to *lock*2) which are lost in the matrix representation. Running the verification tool, we get the fixpoint shown in Figure 7.6. The fixpoint contains 14 elements and is reached after 7 steps. As it does not contain *init*, the mutual exclusion property is verified.

$\{cs1,\ cs1\}$
$\{cs1,\ cs2\}$
$\{cs2,\ cs2\}$
$\{cs2,\ waiting,\ lock1,\ lock2\}$
$\{cs1,\ waiting,\ lock1,\ lock2\}$

Figure 7.7: Fixpoint computation for the mutual exclusion example: first step

We now show how the computation of structural invariants of Section 4.2.2, in combination with the methodology of pruning of Section 6.1.2.2, can be used to optimize the above fixpoint computation. Computing the place invariants for the Petri net in Figure 6.2 yields, among the others, the following properties: for every reachable marking $\mathcal{M}$, $\mathcal{M}(lock1) + \mathcal{M}(cs1) = 1$ and $\mathcal{M}(lock2) + \mathcal{M}(cs2) = 1$ (see for instance [DRB01]). This in turn implies that configurations containing *at least* one token in $lock1$ and one token in $cs1$ (or, respectively, one token in $lock2$ and one token in $cs2$) are unreachable and can therefore be discarded (*pruned*). Using dynamic pruning (this requires a slight modification of the fixpoint operator, see Section 6.1.2.2), this time the verification algorithm converges immediately at the first iteration. In Figure 7.7 we show the partial result computed at the *second* iteration of the algorithm *without* using dynamic pruning. With respect to the initial set of unsafe states, only two elements (the last two) are computed. These elements do not satisfy the above structural invariants, therefore they can be dynamically pruned. The fixpoint consists of the remaining configurations.

Using our verification tool, we have verified that the mutual exclusion property holds for the Petri net in Figure 6.2, independently of the number of initial processes. In other words, we have verified a safety property for a *parametric* system. We conclude this section by showing how even more sophisticated specifications are possible in linear logic. Specifically, we can support *dynamic* generation of *waiting* processes. Let us consider the following modified LO specification for the set of initial states:

$$5'.\ \ init \circ\!\!- demon \otimes lock1 \otimes lock2$$
$$6'.\ \ demon \circ\!\!- demon \otimes waiting$$

Initialization of the system directly creates the two locks and a *demon*. The demon is used to generate waiting processes *at run-time*. In other words, new waiting processes can be dynamically injected in the system at any time after the initialization phase. Evaluating the bottom-up semantics for the LO program obtained by substituting the above clauses 5' and 6' for, respectively, clauses 5 and 6, we can prove that the mutual exclusion property still holds for the above specification. The fixpoint contains 20 elements and is reached after 7 steps. Using dynamic pruning, the fixpoint computation converges immediately in the same way as for the original specification.

7.4 Related Work

The inspiration for our work (see [BDM02]) originally came from decidability results, based on the theory of well-quasi-orderings, for verification algorithms for validation of parameterized systems via a *backward* reachability analysis [ACJT96, AJ98, FS01] (see also Section 6.4). Specifically, in this chapter we have defined a *bottom-up* evaluation algorithm which solves the so-called *covering* problem (see [FS01]) for transition systems defined via the linear logic language LO [AP91b]. The decidability result for propositional LO provability, which we obtain here as a corollary, is not new. In particular, it is consequence of decidability of linear *affine* logic [Kop95]. Given that Petri nets can be encoded in LO (see Section 4.3), our bottom-up evaluation algorithm can be seen as an alternative to the Karp and Miller's *coverability graph* construction for Petri nets [KM69] (which, however, as a difference with ours, is a *forward* construction). Here, we also deal with further connectives like the additive conjunction of linear logic.

From a logic programming perspective, our bottom-up semantics is clearly an extension of the traditional fixpoint semantics for Horn logic [Llo87] to a fragment of linear (affine) logic. As a difference, we need to add special rules, handled via the satisfiability judgment $\Vdash$) to handle formulas built over the logical connectives $\parr$, $\&$, $\top$ and $\bot$.

Other sources of inspiration came from linear logic programming. In [HW98], the authors present an abstract deductive system for the bottom-up evaluation of linear logic programs. The *left introduction*, *weakening* and *cut* rules are used to compute the logical consequences of a given formula. The proof system the authors propose should be thought of as a general scheme to which evaluation of the specific language at hand should conform. The scheme is general enough to allow for different computational strategies, mixing *bottom-up* and *top-down* evaluation, *breadth-first* or *depth-first* search, *eager* or *lazy* evaluation. The satisfiability relations we use in the definition of the fixpoint operators correspond to top-down steps within their bottom-up evaluation scheme. The framework is discussed for a more general fragment than LO, in particular dynamically changing programs are supported. However, they do not provide an *effective* fixpoint operator as we did in the case of LO, and they do not discuss computability issues for the derivability relation.

In [APC97], Andreoli, Pareschi and Castagnetti present a partial evaluation scheme for propositional LO, which works as follows. Given an initial goal G, they use a construction similar to Karp and Miller's *coverability graph* [KM69] for Petri Nets to build a finite representation of a proof tree for G. During the *top-down* construction of the tree for G, they apply in fact a *generalization step* that works as follows. If a goal, say $\mathcal{B}$, that has to be proved is *subsumed* by a node already visited, say $\mathcal{A}$, (i.e., $\mathcal{B} = \mathcal{A} + \mathcal{A}'$), then the part of proof tree between the two goals is replaced by a proof tree for $\mathcal{A} + (\mathcal{A}')^*$, where $\mathcal{A} + (\mathcal{A}')^*$ is a finite representation of the union of $\mathcal{A}$ with the closure of $\mathcal{A}'$. Using Dickson's Lemma, the authors show that the construction always terminates. The main difference with our

approach is that we study a goal independent *bottom-up* evaluation strategy, therefore in our fixpoint semantics we do not need any *generalization* step. In fact, goals which are *subsumed* can be automatically discarded during the bottom-up evaluation process. This simplifies the computation as shown in Example 7.26. The partial evaluation scheme of [APC97] is proposed as a target for analysis of first-order LO programs, via the so-called *counting* abstraction, which consists in abstracting an atom with the corresponding predicate symbol. Clearly, we can do the same with our bottom-up evaluation scheme. However, in this thesis we allow for a much more refined analysis, by discussing directly a non ground semantics for first-order programs, based on unification and most general unifiers (see Chapters 9 and 10).

Finally, we must mention the connection between our semantics for LO and the bottom-up semantics for Disjunctive Logic Programs [MRL91]. In a disjunctive logic program, the head of a clause is a disjunction of atomic formulas, whereas the body is a conjunction of atomic formulas. In the semantics of [MRL91] interpretations are collections of *sets* (i.e., *classical* disjunctions) of atomic formulas, whereas LO interpretations are *multisets* (i.e., $\parr$-disjunctions) of atomic formulas. Therefore, in the propositional case in order to prove the convergence of the fixpoint iteration, we need an argument (Dickson's lemma) stronger than the *finiteness* of the extended Herbrand base of [MRL91] (collection of *minimal sets*). For a detailed discussion about the connection between LO and DLP, we refer the reader to [BDM01b]. Here, we have proved that, from a proof theoretical perspective, DLP can be derived from LO by admitting the structural rule of *contraction*. Furthermore, we have shown that the fixpoint semantics for a DLP program can be obtained by making *abstract interpretation* over the fixpoint semantics of a related LO program (specifically, the abstraction maps any multiset into the corresponding set).

Summary of the Chapter. *In this chapter we have discussed the foundations of a bottom-up semantics for LO specifications. As a first step, we have dealt with propositional programs. We have proved that the bottom-up semantics is correct and complete with respect to the operational semantics, and we have shown an algorithm for evaluating the bottom-up semantics which is always guaranteed to terminate. As an application, we have presented some classical systems, specified in the Petri net formalism, which can be translated into LO theories and validated using this approach.*

Building on the ideas presented here, in the following chapters we will extend the bottom-up semantics to more complex fragments of linear logic. First of all, in the next chapter we will discuss a simple addition enriching propositional LO with a more refined resource management. However, the price to pay will be that provability in general becomes undecidable.

Chapter 8

Refining LO Resource Management: LO with 1

This chapter extends the bottom-up semantics for LO of Chapter 3 by considering a richer set of logical connectives. In fact, the decidability of propositional provability shows that LO is not as interesting as one could expect from a state-oriented extension of the logic programming paradigm. Specifically, LO does not provide a natural way to *count* resources. This feature can be introduced by considering the language $\mathrm{LO}_{\mathbf{1}}$ (see Section 3.3.1), i.e., by considering a slight extension of LO in which we add unit clauses defined via the constant $\mathbf{1}$ and the multiplicative conjunction $\otimes$. The resulting language can be viewed as a first step towards more complex languages based on linear logic like LinLog [And92]. In this chapter we will also show that $\mathrm{LO}_{\mathbf{1}}$ can be used to model more sophisticated models of concurrent systems than LO, like Petri Nets with *transfer arcs* and *broadcast protocols*.

The addition of the constant $\mathbf{1}$ breaks down the decidability of provability which holds instead for propositional LO. If we think about the connection between bottom-up semantics and verification (see Section 6.3), it turns out that allowing axioms of the form $H \circ\!\!- \mathbf{1}$ introduces the possibility of specifying set of configurations (e.g. markings for Petri nets) which are not *upward-closed*. Otherwise said, the result stated in Proposition 3.9 (admissibility of the weakening rule) does not hold anymore for the fragment $\mathrm{LO}_{\mathbf{1}}$. Remembering the connection between LO and Petri nets (see Section 4.3), we have that undecidability of $\mathrm{LO}_{\mathbf{1}}$ provability follows by classical results which show that there is no algorithm to compute the reachability set for Petri nets [EN94]. Despite this negative result, in this chapter we will still be able to define an *effective* symbolic fixpoint operator for $\mathrm{LO}_{\mathbf{1}}$.

Technically, we proceed as in Chapter 7. In particular, we formulate the bottom-up evaluation procedure in two steps. We first present a simple, non-effective notion of (concrete) interpretations and the corresponding definition of fixpoint operator. Then, we define an

$$\frac{}{P \vdash_{\mathbf{1}} \top, \Delta}\ \top_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, G_2, \Delta}{P \vdash_{\mathbf{1}} G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, \Delta \quad P \vdash_{\mathbf{1}} G_2, \Delta}{P \vdash_{\mathbf{1}} G_1 \,\&\, G_2, \Delta}\ \&_r \qquad \frac{P \vdash_{\mathbf{1}} \Delta}{P \vdash_{\mathbf{1}} \bot, \Delta}\ \bot_r$$

$$\frac{}{P \vdash_{\mathbf{1}} \mathbf{1}}\ \mathbf{1}_r \qquad \frac{P \vdash_{\mathbf{1}} G_1, \Delta_1 \quad P \vdash_{\mathbf{1}} G_2, \Delta_2}{P \vdash_{\mathbf{1}} G_1 \otimes G_2, \Delta_1, \Delta_2}\ \otimes_r \qquad \frac{P \vdash_{\mathbf{1}} G, \mathcal{A}}{P \vdash_{\mathbf{1}} \widehat{H}, \mathcal{A}}\ bc\ \ (H \circ\!\!- G\ \in P)$$

Figure 8.1: A proof system for propositional $\mathrm{LO}_{\mathbf{1}}$

abstract domain and a new symbolic fixpoint operator, which is shown to be equivalent to the previous one. The abstract domain is a *constraint system* in the sense of [AJ01b], i.e., it provides *assertions* to symbolically represent infinite sets of states. Specifically, we will introduce an abstract domain based on a special class of *linear constraints* defined over integer variables that *count* resources. This abstract domain generalizes the domain used for propositional LO: the latter can be represented as the subclass of constraints with no equalities. We remark that the use of constraints we make in this chapter is radically different from that of Chapter 9. Here, we use constraints *at the meta-level* as symbolic representations of *semantical* objects, while in Chapter 9 constraints are introduced at the *syntactical* (and also semantical) level, as a means to allow for reasoning on specialized and heterogeneous domains (e.g. integers, reals, strings, and so on) inside LO specifications.

For the sake of simplicity, we will overload some notations used in Chapter 7. Namely, the notations $O(P), F(P)$ for operational and fixpoint semantics, T_P, S_P for the concrete and symbolic fixpoint operators, and $\models, \Vdash$ for the concrete and abstract satisfiability judgments will be re-used to denote the analogous concepts for the language $\mathrm{LO}_{\mathbf{1}}$.

This chapter extends our previous works [BDM00, BDM02].

8.1 A Bottom-Up Semantics for $\mathrm{LO}_{\mathbf{1}}$

In the following we will use the same notations introduced in Chapter 7. We always assume a fixed signature Σ including a finite set of propositional symbols $a_1, \ldots, a_n$. For the convenience of the reader, in Figure 8.1 we recall a proof system for propositional $\mathrm{LO}_{\mathbf{1}}$ (see also Section 3.3.1). A sequent is now provable if all branches of its proof tree terminate with instances of the $\top_r$ or the $\mathbf{1}_r$ axiom.

The top-down operational semantics can be defined as follows.

Definition 8.1 (Operational Semantics) *Given a propositional $LO_{\mathbf{1}}$ program P, its*

operational semantics, denoted $O(P)$, is given by

$$O(P) = \{\mathcal{A} \mid \mathcal{A} \textit{ is a fact and } P \vdash_{\mathbf{1}} \mathcal{A}\}.$$

We first note that, in contrast with Proposition 3.9, the weakening rule is not admissible in $\mathrm{LO}_{\mathbf{1}}$. This implies that we cannot use the same techniques we used for LO for defining the abstract domain and also for proving termination of the bottom-up evaluation. So the question is: can we still find a finite representation of $O(P)$? The following proposition gives us a negative answer.

Proposition 8.2 (Undecidability of $\mathrm{LO}_{\mathbf{1}}$ provability) *Given a propositional $LO_{\mathbf{1}}$ program P, there is no algorithm to compute $O(P)$.*

Proof We prove the proposition by presenting an encoding of Petri nets into $\mathrm{LO}_{\mathbf{1}}$. Let a Petri net N, with a set of initial markings $I = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$, be given. Consider the LO (and thus $\mathrm{LO}_{\mathbf{1}}$) propositional program $P = \mathcal{P}(R(N))$ (see Section 4.3). Furthermore, for every marking $\mathcal{M}_i \in I$ consider the $\mathrm{LO}_{\mathbf{1}}$ clause $H_i \circ\!\!- \mathbf{1}$, where H_i is a $\parr$-disjunction of propositional symbols such that $\widehat{H_i} = \mathcal{M}_i$, for $i : 1, \ldots, n$. Let P' be the program consisting of P and the above n clauses. Now, it is easy to recognize that computing the operational semantics of P' amounts to computing the reachability set of N with initial markings I. From classical results on Petri Nets (see e.g. the survey [EN94]), there is no algorithm to compute this set. The conclusion follows by reduction to the *marking equivalence* problem that is known to be undecidable. □

As the reader can see, the above encoding of Petri nets into $\mathrm{LO}_{\mathbf{1}}$ does not make use of the multiplicative conjunction $\otimes$, therefore the operational semantics for the fragment of $\mathrm{LO}_{\mathbf{1}}$ *without* $\otimes$ (i.e., LO with the constant $\mathbf{1}$) is still not computable. Despite Proposition 8.2, it is still possible to define a *symbolic*, effective fixpoint operator for $\mathrm{LO}_{\mathbf{1}}$ programs, as shown in Section 8.2. Before going into more details, we first rephrase the semantics of Section 7.1 for $\mathrm{LO}_{\mathbf{1}}$. We omit the relevant proofs, which are analogous to those of Section 7.1. The definitions of Herbrand base and interpretations are the same as in Chapter 7. The concrete semantics is defined on the usual domain $\langle \mathcal{D}, \subseteq \rangle$, where $\mathcal{D} = \mathcal{P}(B_P)$.

Definition 8.3 (Satisfiability Judgment) *Let P be a propositional $LO_{\mathbf{1}}$ program and I*

an interpretation. The satisfiability judgment $\models$ *is defined as follows:*

$I \models \top, \Delta \blacktriangleright \mathcal{A}'$ *for any fact* $\mathcal{A}'$;
$I \models \mathbf{1} \blacktriangleright \epsilon$;
$I \models \mathcal{A} \blacktriangleright \mathcal{A}'$ *if* $\mathcal{A} + \mathcal{A}' \in I$;
$I \models G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{A}$ *if* $I \models G_1, \Delta \blacktriangleright \mathcal{A}$ *and* $I \models G_2, \Delta \blacktriangleright \mathcal{A}$;
$I \models G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \mathcal{A}_1 + \mathcal{A}_2$ *if* $I \models G_1, \Delta_1 \blacktriangleright \mathcal{A}_1$ *and* $I \models G_2, \Delta_2 \blacktriangleright \mathcal{A}_2$;
$I \models G_1 \parr G_2, \Delta \blacktriangleright \mathcal{A}$ *if* $I \models G_1, G_2, \Delta \blacktriangleright \mathcal{A}$;
$I \models \bot, \Delta \blacktriangleright \mathcal{A}$ *if* $I \models \Delta \blacktriangleright \mathcal{A}$.

The satisfiability judgment $\models$ satisfies the following properties.

Lemma 8.4 *For every interpretation* I*, context* Δ *and fact* $\mathcal{A}$*,*

$$I \models \Delta \blacktriangleright \mathcal{A} \text{ iff } I \models \Delta, \mathcal{A} \blacktriangleright \epsilon.$$

Lemma 8.5 *For any interpretations* I_1, I_2*, context* Δ*, and fact* $\mathcal{A}$*,*

i. if $I_1 \subseteq I_2$ *and* $I_1 \models \Delta \blacktriangleright \mathcal{A}$ *then* $I_2 \models \Delta \blacktriangleright \mathcal{A}$*;*

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ *and* $\bigcup_{i=1}^{\infty} I_i \models \Delta \blacktriangleright \mathcal{A}$ *then there exists* $k \in \mathbb{N}$ *s.t.* $I_k \models \Delta \blacktriangleright \mathcal{A}$.

The fixpoint operator T_P is defined like the one for LO.

Definition 8.6 (Fixpoint Operator T_P) *Given a propositional* $LO_{\mathbf{1}}$ *program* P *and an interpretation* I*, the fixpoint operator* T_P *is defined as follows:*

$$T_P(I) = \{\widehat{H} + \mathcal{A} \mid H \circ\!\!- G \in P,\ I \models G \blacktriangleright \mathcal{A}\}.$$

The following property holds.

Proposition 8.7 (Monotonicity and Continuity) *For every propositional* $LO_{\mathbf{1}}$ *program* P*, the fixpoint operator* T_P *is monotonic and continuous over the lattice* $\langle \mathcal{D}, \subseteq \rangle$.

Monotonicity and continuity of the T_P operator imply, by Tarski's Theorem, that $lfp(T_P) = T_P{\uparrow_\omega}$. The fixpoint semantics of a propositional $\mathrm{LO}_{\mathbf{1}}$ program P is then defined as follows.

Definition 8.8 (Fixpoint Semantics) *Given a propositional LO_1 program P, its fixpoint semantics, denoted $F(P)$, is defined as follows:*

$$F(P) = lfp(T_P) = T_P{\uparrow_\omega} .$$

The fixpoint semantics is sound and complete with respect to the operational semantics, as stated in the following theorem.

Theorem 8.9 (Soundness and Completeness) *For every propositional LO_1 program P, $F(P) = O(P)$.*

8.2 Constraint Semantics for $\mathbf{LO_1}$

In this section we will introduce an abstract domain (a constraint system using the terminology of [AJ01b]) which consists of constraint-based representations of provable multisets, and we will define a *symbolic* fixpoint operator over this domain. The application of this operator is effective. Proposition 8.2 shows however that there is no guarantee that its fixpoint can be reached after finitely many steps. Using the analogy between Petri nets and *vector addition systems*, we make use of a class of constraints over integer variables counting occurrences of atoms inside multisets (markings). Specifically, we use vectors of variables $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, where variable x_i denotes the number of occurrences of $a_i \in \Sigma$ in a given fact. We give the following definition.

Definition 8.10 (Occurrence Constraints) *Let $\mathcal{A}$ be a fact. We define the following class of constraints.*

$$\begin{aligned}
&\alpha_{\mathcal{A}}(\mathbf{x}, \mathbf{x}') \;\equiv\; \textstyle\bigwedge_{i=1}^{n} x_i = x_i' + \mathcal{A}(a_i) \\
&\rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}') \;\equiv\; \textstyle\bigwedge_{i=1}^{n} x_i = x_i' - \mathcal{A}(a_i) \;\wedge\; x_i \geq 0 \\
&\phi_{\mathcal{A}} \;\equiv\; \textstyle\bigwedge_{i=1}^{n} x_i = \mathcal{A}(a_i) \\
&\phi_{[\![\mathcal{A}]\!]} \;\equiv\; \textstyle\bigwedge_{i=1}^{n} x_i \geq \mathcal{A}(a_i) \\
&Rem(\varphi, \mathcal{A}) = \exists \mathbf{x}'.(\varphi[\mathbf{x}'/\mathbf{x}] \wedge \rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}')) \\
&Add(\varphi, \mathcal{A}) = \exists \mathbf{x}'.(\varphi[\mathbf{x}'/\mathbf{x}] \wedge \alpha_{\mathcal{A}}(\mathbf{x}, \mathbf{x}')) \\
&\varphi_1 \otimes \varphi_2 = \exists \mathbf{x}'.\exists \mathbf{x}''.(\varphi_1[\mathbf{x}'/\mathbf{x}] \wedge \varphi_2[\mathbf{x}''/\mathbf{x}] \wedge \mathbf{x} = \mathbf{x}' + \mathbf{x}'')
\end{aligned}$$

Example 8.11 Let $\Sigma = \{a_1, a_2\}$, and consider the multisets $\mathcal{A} = \{a_1, a_1, a_2\}$, $\mathcal{B} = \{a_1, a_2\}$. Then, $\phi_{\mathcal{A}} \equiv x_1 = 2 \wedge x_2 = 1$, $\phi_{[\![\mathcal{A}]\!]} \equiv x_1 \geq 2 \wedge x_2 \geq 1$, $Rem(\phi_{[\![\mathcal{A}]\!]}, \mathcal{B}) \equiv$

$\exists x'_1, x'_2.(x'_1 \geq 2 \wedge x'_2 \geq 1 \wedge x_1 = x'_1 - 1 \wedge x_2 = x'_2 - 1 \wedge x_1 \geq 0 \wedge x_2 \geq 0)$. This latter constraint is equivalent to $x_1 \geq 1 \wedge x_2 \geq 0$. We also have that $\phi_{[\![\mathcal{A}]\!]} \otimes \phi_{\mathcal{B}} \equiv \exists x'_1, x'_2, x''_1, x''_2.(x'_1 \geq 2 \wedge x'_2 \geq 1 \wedge x''_1 = 1 \wedge x''_2 = 1 \wedge x_1 = x'_1 + x''_1 \wedge x_2 = x'_2 + x''_2)$. This constraint is equivalent to $x_1 \geq 3 \wedge x_2 \geq 2$. □

Using constraints of the form $\phi_{[\![\mathcal{A}]\!]}$, for a fact $\mathcal{A}$, we can immediately recover the semantics of Section 7.2. All the operations on multisets involved in the definition of S_P (see Definition 7.14) can be expressed as operations over linear constraints. For instance, the ideal generated by the empty multiset ϵ corresponds to the linear constraint $\phi_{[\![\epsilon]\!]}$; given the ideals $[\![\mathcal{A}]\!]$ and $[\![\mathcal{B}]\!]$, the ideal $[\![\mathcal{A} \bullet \mathcal{B}]\!]$ is represented as the constraint $\phi_{[\![\mathcal{A}]\!]} \wedge \phi_{[\![\mathcal{B}]\!]}$, $[\![\mathcal{B} \backslash \mathcal{A}]\!]$ is represented as the constraint $Rem(\phi_{[\![\mathcal{B}]\!]}, \mathcal{A})$. The constraint $\rho_{\mathcal{A}}$ models the removal of the occurrences of the propositional symbols in $\mathcal{A}$ from all elements of the denotation of $\mathcal{B}$. Similarly, $[\![\mathcal{B} + \mathcal{A}]\!]$, for a given multiset $\mathcal{A}$, is represented as the constraint $Add(\phi_{[\![\mathcal{B}]\!]}, \mathcal{A})$.

The introduction of the constant $\mathbf{1}$ breaks down Proposition 3.9. As a consequence, the abstraction based on ideals is not precise anymore. In order to give a semantics for $\mathrm{LO}_{\mathbf{1}}$, we need to add a class of constraints for representing collections of multisets that are not upward-closed. Specifically, we need equality constraints of the form $\phi_{\mathcal{A}}$. The operations over linear constraints discussed previously extend smoothly when adding this new class of equality constraints. In particular, given two constraints φ_1 and φ_2, their conjunction $\varphi_1 \wedge \varphi_2$ still plays the role that the operation $\bullet$ (least upper bound of multisets) had in Definition 7.14, while $Rem(\varphi, \mathcal{A})$, for a given constraint φ and a multiset $\mathcal{A}$, plays the role of multiset difference. The reader can compare Definition 7.14 with Definition 8.15.

For the class of linear integer constraints presented above, it is well-known that there are algorithms for checking satisfiability, entailment, and for variable elimination (see e.g. [BGP97]).

Based on these ideas, we can define a bottom-up evaluation procedure for $\mathrm{LO}_{\mathbf{1}}$ programs via a symbolic operator S_P. In the following we will use the notation $\widehat{\mathbf{c}}$, where $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$ is a solution of a constraint φ (i.e., an assignment of natural numbers to the variables $\mathbf{x}$ which satisfies φ), to indicate the multiset over $\Sigma = \{a_1, \ldots, a_n\}$ which contains c_i occurrences of every propositional symbol a_i (formally, $\widehat{\mathbf{c}} = c_1 \cdot \{a_1\} + \ldots + c_n \cdot \{a_n\}$). Note that according to this notation, $\mathbf{c}$ is the unique solution of $\phi_{\widehat{\mathbf{c}}}$ (e.g. $\langle 1, 0, 0, \ldots \rangle$ is the unique solution of $\phi_{\{a_1\}}$). We extend this definition to a set C of constraint solutions by $\widehat{C} = \{\widehat{\mathbf{c}} \mid \mathbf{c} \in C\}$. We then define the denotation of a given constraint φ, written $[\![\varphi]\!]$, as the set of multisets corresponding to solutions of φ. Formally, we have the following definition.

Definition 8.12 (Denotation of a Constraint) *Given a linear constraint φ over the integer variables $\langle x_1, \ldots, x_n \rangle$, its denotation $[\![\phi]\!]$ is given by*

$$[\![\varphi]\!] = \{\widehat{\mathbf{c}} \mid \mathbf{x} = \mathbf{c} \textit{ satisfies } \varphi\}.$$

Two constraints φ and ψ are said to be equivalent, written $\varphi \simeq \psi$, if and only if $[\![\varphi]\!] = [\![\psi]\!]$ (i.e., we identify constraints with the same set of solutions).

For the sake of simplicity, in the following we will identify a constraint with its equivalence class, i.e., we will simply write φ instead of $[\varphi]_{\simeq}$. Let LC_Σ be the set of (equivalence classes of) of linear constraints over the integer variables $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ associated to the signature $\Sigma = \{a_1, \ldots, a_n\}$. The operator S_P is defined on *constraint interpretations* consisting of sets (disjunctions) of (equivalence classes of) linear constraints. We have the following definitions.

Definition 8.13 (Constraint Interpretation) *We say that $I \subseteq LC_\Sigma$ is a constraint interpretation. Constraint interpretations form a complete lattice $\langle \mathcal{D}, \subseteq \rangle$ with respect to set inclusion, where $\mathcal{D} = \mathcal{P}(LC_\Sigma)$.*

Definition 8.14 (Denotation of a Constraint Interpretation) *Given a constraint interpretation I, its denotation $[\![I]\!]$ is the interpretation given by*

$$[\![I]\!] = \{[\![\varphi]\!] \mid \varphi \in I\}.$$

For brevity, we will define the semantics directly on interpretations consisting of the representative elements of the equivalence classes. Also, in the following definitions we assume that the conditions apply only when the constraints involved are *satisfiable*. First of all, we extend the definition of the satisfiability judgment using operations over constraints (see Definition 8.10) as follows.

Definition 8.15 (Abstract Satisfiability Judgment) *Let P be a propositional $LO_{\mathbf{1}}$ program and I an interpretation. The abstract satisfiability judgment $\Vdash$ is defined as follows:*

$$I \Vdash \top, \Delta \blacktriangleright \phi_{[\![\epsilon]\!]};$$

$$I \Vdash \mathbf{1} \blacktriangleright \phi_\epsilon;$$

$$I \Vdash \mathcal{A} \blacktriangleright Rem(\psi, \mathcal{A}), \ \textit{for } \psi \in I;$$

$$I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \varphi_1 \wedge \varphi_2 \ \textit{if } I \Vdash G_1, \Delta \blacktriangleright \varphi_1, \ I \Vdash G_2, \Delta \blacktriangleright \varphi_2;$$

$$I \Vdash G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \varphi_1 \otimes \varphi_2 \ \textit{if } I \Vdash G_1, \Delta_1 \blacktriangleright \varphi_1, \ I \Vdash G_2, \Delta_2 \blacktriangleright \varphi_2;$$

$$I \Vdash G_1 \parr G_2, \Delta \blacktriangleright \varphi \ \textit{if } I \Vdash G_1, G_2, \Delta \blacktriangleright \varphi;$$

$$I \Vdash \bot, \Delta \blacktriangleright \varphi \ \textit{if } I \Vdash \Delta \blacktriangleright \varphi.$$

The connection between the satisfiability judgments $\models$ and $\Vdash$ is clarified by the following lemma.

Lemma 8.16 *For every interpretation I, context Δ, and constraint φ,*

i. if $I \Vdash \Delta \blacktriangleright \varphi$, then $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}$ for every $\mathcal{A} \in [\![\varphi]\!]$;

ii. if $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}$, then there exists φ such that $I \Vdash \Delta \blacktriangleright \varphi$ and $\mathcal{A} \in [\![\varphi]\!]$.

Proof

i. By induction on the derivation of $I \Vdash \Delta \blacktriangleright \varphi$.

- If $I \Vdash \top, \Delta \blacktriangleright \varphi$, then every $\mathbf{c}$ (with $c_i \geq 0$) is solution of $\varphi \equiv \bigwedge_{i=1}^{n} x_i \geq 0$, and $[\![I]\!] \models \top, \Delta \blacktriangleright \mathcal{A}'$ for every fact $\mathcal{A}'$;
- if $I \Vdash \mathbf{1} \blacktriangleright \varphi$, then $\langle 0, \ldots, 0 \rangle$ is the only solution of φ, and $[\![I]\!] \models \mathbf{1} \blacktriangleright \epsilon$;
- if $I \Vdash \mathcal{A} \blacktriangleright \varphi$ then there exists $\psi \in I$ s.t. $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}'))$ is satisfiable. Then for every solution $\mathbf{c}$ of φ there exists a vector $\mathbf{c}'$ s.t. $\psi[\mathbf{c}'/\mathbf{x}]$ is satisfiable and $c'_1 \geq \mathcal{A}(a_1), c_1 = c'_1 - \mathcal{A}(a_1), \ldots, c'_n \geq \mathcal{A}(a_n), c_n = c'_n - \mathcal{A}(a_n)$. From this we get that for $i = 1, \ldots, n$, $c'_i = c_i + \mathcal{A}(a_i)$ is a solution for ψ, therefore $\widehat{\mathbf{c}} + \mathcal{A} \in [\![\psi]\!] \subseteq [\![I]\!]$ so that we can conclude $[\![I]\!] \models \mathcal{A} \blacktriangleright \widehat{\mathbf{c}}$;
- if $I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \varphi$ then $\varphi \equiv \varphi_1 \wedge \varphi_2$ and $I \Vdash G_1, \Delta \blacktriangleright \varphi_1$, $I \Vdash G_2, \Delta \blacktriangleright \varphi_2$. By inductive hypothesis, $[\![I]\!] \models G_1, \Delta \blacktriangleright \widehat{\mathbf{c_1}}$ and $[\![I]\!] \models G_2, \Delta \blacktriangleright \widehat{\mathbf{c_2}}$ for every $\mathbf{c_1}$ and $\mathbf{c_2}$ solutions of φ_1 and φ_2, respectively. Thus $[\![I]\!] \models G_1 \,\&\, G_2, \Delta \blacktriangleright \widehat{\mathbf{c}}$ for every $\mathbf{c}$ which is solution of both φ_1 and φ_2, i.e., for every $\mathbf{c}$ which is solution of $\varphi_1 \wedge \varphi_2$;
- if $I \Vdash G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \varphi$ then then $\varphi = \varphi_1 \otimes \varphi_2$ and $I \Vdash G_1, \Delta_1 \blacktriangleright \varphi_1$, $I \Vdash G_2, \Delta_2 \blacktriangleright \varphi_2$. By inductive hypothesis, $[\![I]\!] \models G_1, \Delta_1 \blacktriangleright \widehat{\mathbf{c_1}}$ and $[\![I]\!] \models G_2, \Delta_2 \blacktriangleright \widehat{\mathbf{c_2}}$ for every $\mathbf{c_1}$ and $\mathbf{c_2}$ solutions of φ_1 and φ_2, respectively. Hence, we get that $[\![I]\!] \models G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \widehat{\mathbf{c_1}} + \widehat{\mathbf{c_2}}$ for every $\mathbf{c_1}$ and $\mathbf{c_2}$ solutions of φ_1 and φ_2, i.e., $[\![I]\!] \models G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \widehat{\mathbf{c}}$ for every $\mathbf{c}$ which is solution of $\varphi_1 \otimes \varphi_2$;
- the $\parr$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[\![I]\!] \models \Delta \blacktriangleright \mathcal{A}$.

- $[\![I]\!] \models \top, \Delta \blacktriangleright \widehat{\mathbf{c}}$ for every $\mathbf{c}$, and $I \Vdash \top, \Delta \blacktriangleright \varphi$, where $\varphi \equiv x_1 \geq 0, \ldots, x_n \geq 0$, and every $\mathbf{c}$ is solution of φ;
- if $[\![I]\!] \models \mathbf{1} \blacktriangleright \epsilon$, $\epsilon = \widehat{\langle 0, \ldots, 0 \rangle}$, then $I \Vdash \mathbf{1} \blacktriangleright \varphi$, where $\varphi \equiv x_1 = 0, \ldots, x_n = 0$, and $\langle 0, \ldots, 0 \rangle$ is solution of φ;
- if $[\![I]\!] \models \mathcal{A} \blacktriangleright \widehat{\mathbf{c}}$ then $\widehat{\mathbf{c}} + \mathcal{A} \in [\![I]\!]$, therefore there exists $\psi \in I$ s.t. $\widehat{\mathbf{c}} + \mathcal{A} \in [\![\psi]\!]$. Therefore, if $\mathbf{a}$ is such that $\widehat{\mathbf{a}} = \mathcal{A}$, we have that $\psi[\mathbf{c} + \mathbf{a}/x]$ is satisfiable, $\mathbf{c}$ is solution of $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}'))$ and $I \Vdash \mathcal{A} \blacktriangleright \varphi$;

- if $[\![I]\!] \models G_1 \,\&\, G_2, \Delta \blacktriangleright \widehat{\mathbf{c}}$ then $[\![I]\!] \models G_1, \Delta \blacktriangleright \widehat{\mathbf{c}}$ and $[\![I]\!] \models G_2, \Delta \blacktriangleright \widehat{\mathbf{c}}$. By inductive hypothesis, there exist φ_1 and φ_2 such that $I \Vdash G_1, \Delta \blacktriangleright \varphi_1$ and $I \Vdash G_2, \Delta \blacktriangleright \varphi_2$, and $\mathbf{c}$ is a solution of φ_1 and φ_2. Therefore $I \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \varphi_1 \wedge \varphi_2$ and $\mathbf{c}$ is a solution of $\varphi_1 \wedge \varphi_2$;
- if $[\![I]\!] \models G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \widehat{\mathbf{c}}$ then there exist $\mathbf{c_1}$ and $\mathbf{c_2}$ s.t. $\mathbf{c} = \mathbf{c_1} + \mathbf{c_2}$, $[\![I]\!] \models G_1, \Delta_1 \blacktriangleright \widehat{\mathbf{c_1}}$ and $[\![I]\!] \models G_2, \Delta_2 \blacktriangleright \widehat{\mathbf{c_2}}$. By inductive hypothesis, there exist φ_1 and φ_2 such that $I \Vdash G_1, \Delta_1 \blacktriangleright \varphi_1$ and $I \Vdash G_2, \Delta_2 \blacktriangleright \varphi_2$, $\mathbf{c_1}$ is a solution of φ_1 and $\mathbf{c_2}$ is a solution of φ_2. Therefore $I \Vdash G_1 \otimes G_2, \Delta_1, \Delta_2 \blacktriangleright \varphi_1 \otimes \varphi_2$ and $\mathbf{c}$ is a solution of $\varphi_1 \otimes \varphi_2$;
- the ⅋-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

□

The satisfiability judgment $\Vdash$ also satisfies the following properties.

Lemma 8.17 *For every interpretation $I_1, I_2, \ldots$, context Δ, and constraint φ,*

i. if $I_1 \subseteq I_2$ and $I_1 \Vdash \Delta \blacktriangleright \varphi$, then $I_2 \Vdash \Delta \blacktriangleright \varphi$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^{\infty} I_i \Vdash \Delta \blacktriangleright \varphi$ then there exists $k \in \mathbb{N}$ s.t. $I_k \Vdash \Delta \blacktriangleright \varphi$.

Proof

i. By simple induction on the derivation of $I_1 \Vdash \Delta \blacktriangleright \varphi$.

ii. By induction on the derivation of $\bigcup_{i=1}^{\infty} I_i \Vdash \Delta \blacktriangleright \varphi$.

- The $\top$ and $\mathbf{1}$-cases follow by definition;
- if $\bigcup_{i=1}^{\infty} I_i \Vdash \mathcal{A} \blacktriangleright \varphi$, then there exists $\psi \in \bigcup_{i=1}^{\infty} I_i$ s.t. $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}'))$ is satisfiable. Then there exists k s.t. $\psi \in I_k$ and $I_k \Vdash \mathcal{A} \blacktriangleright \varphi$;
- if $\bigcup_{i=1}^{\infty} I_i \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \varphi$, then $\varphi \equiv \varphi_1 \wedge \varphi_2$, and, by inductive hypothesis, there exist k_1 and k_2 s.t. $I_{k_1} \Vdash G_1, \Delta \blacktriangleright \varphi_1$ and $I_{k_2} \Vdash G_2, \Delta \blacktriangleright \varphi_2$. Then, for $k = max\{k_1, k_2\}$, we have, by i, $I_k \Vdash G_1, \Delta \blacktriangleright \varphi_1$ and $I_k \Vdash G_2, \Delta \blacktriangleright \varphi_2$, therefore $I_k \Vdash G_1 \,\&\, G_2, \Delta \blacktriangleright \varphi_1 \wedge \varphi_2$;
- the $\otimes$-case is analogous to the $\&$-case above;
- the ⅋-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

□

We are now ready to define the extended operator S_P.

Definition 8.18 (Abstract Fixpoint Operator S_P) *Given a propositional LO_1 program P and an interpretation I, the abstract fixpoint operator S_P is defined as follows:*

$$S_P(I) = \{Add(\psi, \widehat{H}) \mid H \multimap G \in P,\ I \Vdash G \blacktriangleright \psi\}$$

We recall that the constraint $Add(\cdot,\cdot)$ is defined in Definition 8.10. The following property shows that S_P is sound and complete w.r.t. T_P.

Proposition 8.19 *For every propositional LO_1 program P and interpretation I, $[\![S_P(I)]\!]$ $= T_P([\![I]\!])$.*

Proof Let $\widehat{\mathbf{c}} \in [\![S_P(I)]\!]$, then there exist $\varphi \in S_P(I)$ and a clause $H \multimap G \in P$, s.t. $\mathbf{c}$ is solution of φ, $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \alpha_{\widehat{H}}(\mathbf{x}, \mathbf{x}'))$ and $I \Vdash G \blacktriangleright \psi$. Then there exists $\mathbf{c}'$ solution of ψ s.t. $\widehat{\mathbf{c}} = \widehat{\mathbf{c}'} + \widehat{H}$, and, by Lemma 8.16 i, $[\![I]\!] \models G \blacktriangleright \widehat{\mathbf{c}'}$. Therefore, by definition of T_P, $\widehat{\mathbf{c}} = \widehat{\mathbf{c}'} + \widehat{H} \in T_P([\![I]\!])$.
Vice versa, let $\widehat{\mathbf{c}} \in T_P([\![I]\!])$, then there exists $H \multimap G \in P$ s.t. $[\![I]\!] \models G \blacktriangleright \mathcal{A}$ and $\widehat{\mathbf{c}} = \widehat{H} + \mathcal{A}$. By Lemma 8.16 ii, there exists ψ s.t. $I \Vdash G \blacktriangleright \psi$ and $\mathcal{A} \in [\![\psi]\!]$. Therefore $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \alpha_{\widehat{H}}(\mathbf{x}, \mathbf{x}')) \in S_P(I)$, and $\widehat{\mathbf{c}} = \widehat{H} + \mathcal{A} \in [\![\varphi]\!] \subseteq [\![S_P(I)]\!]$. □

The abstract fixpoint operator S_P satisfies the following property.

Proposition 8.20 (Monotonicity and Continuity) *For every propositional LO_1 program P, the abstract fixpoint operator S_P is monotonic and continuous over the lattice $\langle \mathcal{D}, \subseteq \rangle$.*

Proof *Monotonicity.* Immediate from S_P definition and Lemma 8.17 i.
Continuity. Let $I_1 \subseteq I_2 \subseteq \ldots$, be a chain of interpretations. We show that $S_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} S_P(I_i)$. If $\varphi \in S_P(\bigcup_{i=1}^{\infty} I_i)$, by definition there exists a clause $H \multimap G \in P$ s.t. $\bigcup_{i=1}^{\infty} I_i \Vdash G \blacktriangleright \psi$ and $\varphi \equiv \exists \mathbf{x}'.(\psi[\mathbf{x}'/\mathbf{x}] \wedge \alpha_{\widehat{H}}(\mathbf{x}, \mathbf{x}'))$ is satisfiable. By Lemma 8.17 ii, there exists k s.t. $I_k \Vdash G \blacktriangleright \psi$. This implies that $\varphi \in S_P(I_k)$, i.e., $\varphi \in \bigcup_{i=1}^{\infty} S_P(I_i)$. □

Corollary 8.21 $[\![lfp(S_P)]\!] = lfp(T_P)$.

Now, let $\mathcal{F}_{sym}(P) = lfp(S_P)$, then we have the following main theorem. which shows that S_P can be used (without termination guarantee) to compute symbolically the set of logical consequences of an $\mathrm{LO}_\mathbf{1}$ program.

Theorem 8.22 (Soundness and Completeness) *For every propositional $LO_{\mathbf{1}}$ program P, $O(P) = F(P) = [\![\mathcal{F}_{sym}(P)]\!]$.*

Proof By Theorem 8.9 and Corollary 8.21. □

8.3 Bottom-Up Evaluation for $\mathrm{LO}_{\mathbf{1}}$

Using a constraint-based representation for $\mathrm{LO}_{\mathbf{1}}$ provable multisets, we have reduced the problem of computing $O(P)$ to the problem of computing the reachable states of a system with *integer* variables. As shown by Proposition 8.2, the termination of the algorithm is not guaranteed a priori. In this respect, Theorem 7.24 gives us sufficient conditions that ensure its termination. The abstract fixpoint operator S_P of Section 8.2 is defined over the lattice $\langle \mathcal{P}(LC_\Sigma), \subseteq \rangle$, with set inclusion being the partial order relation and set union the least upper bound operator. When we come to a concrete implementation of S_P, it is worth considering a weaker ordering relation between interpretations, namely pointwise *subsumption*. By analogy with the multiset inclusion relation $\preccurlyeq$, let $\preccurlyeq^c$ be the partial order between (equivalence classes of) constraints given by $\varphi \preccurlyeq^c \psi$ if and only if $[\![\psi]\!] \subseteq [\![\varphi]\!]$. In accordance with the definition of pointwise subsumption given in Section 6.1.2, we say that an interpretation I is subsumed by an interpretation J, written $I \sqsubseteq J$, if and only if for every $\varphi \in I$ there exists $\psi \in J$ such that $\psi \preccurlyeq^c \varphi$.

As we do not need to distinguish between different interpretations representing the same set of solutions, we can consider interpretations I and J to be equivalent in case both $I \sqsubseteq J$ and $J \sqsubseteq I$ hold. In this way, we get a lattice of interpretations ordered by $\sqsubseteq$ and such that the least upper bound operator is still set union. This construction is the natural extension of the one of Section 7.2. Actually, when we limit ourselves to considering LO programs (i.e., without the constant $\mathbf{1}$) it turns out that we need only consider constraints of the form $\mathbf{x} \geq \mathbf{c}$, which can be abstracted away by considering the upward closure of $\widehat{\mathbf{c}}$, as we did in Section 7.2. The reader can note that the $\preccurlyeq^c$ relation defined above for constraints is an extension of the multiset inclusion relation we used in Section 7.2.

The construction based on $\sqsubseteq$ can be directly incorporated into the semantic framework presented in Section 8.2, where, for the sake of simplicity, we have adopted an approach based on $\subseteq$. Of course, relation $\subseteq$ is stronger than $\sqsubseteq$, therefore a computation based on $\sqsubseteq$ is correct and it terminates every time a computation based on $\subseteq$ does. However, the converse does not always hold, and this is why a concrete algorithm for computing the least fixpoint of S_P relies on subsumption. Let us see an example.

Example 8.23 We calculate the fixpoint semantics for the following $\mathrm{LO}_{\mathbf{1}}$ program made

$$
\begin{array}{rcll}
S_P{\uparrow}_1 & = & \{ & x_a = 1 \wedge x_b = 0 \wedge x_c = 0, \quad x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0, \\
 & & & x_a \geq 0 \wedge x_b \geq 0 \wedge x_c \geq 2 \quad \} \\
 & & & \\
S_P{\uparrow}_2 & = & \{ & x_a = 0 \wedge x_b = 2 \wedge x_c = 0, \quad x_a \geq 0 \wedge x_b \geq 3 \wedge x_c \geq 0, \\
 & & & x_a \geq 2 \wedge x_b \geq 0 \wedge x_c \geq 0 \quad \} \ \cup \ S_P{\uparrow}_1 \\
 & & & \\
S_P{\uparrow}_3 & = & \{ & x_a = 0 \wedge x_b = 1 \wedge x_c = 1, \quad x_a \geq 0 \wedge x_b \geq 2 \wedge x_c \geq 1, \\
 & & & x_a \geq 1 \wedge x_b \geq 0 \wedge x_c \geq 1 \quad \} \ \cup \ S_P{\uparrow}_2
\end{array}
$$

Figure 8.2: Symbolic fixpoint computation for an $\mathrm{LO}_{\mathbf{1}}$ program

up of six clauses:

1. $a \multimap \mathbf{1}$
2. $a \parr b \multimap \top$
3. $c \parr c \multimap \top$
4. $b \parr b \multimap a$
5. $a \multimap b$
6. $c \multimap a \,\&\, b$

Let $\Sigma = \{a, b, c\}$ and consider constraints over the variables $\mathbf{x} = \langle x_a, x_b, x_c \rangle$. We have that $S_P{\uparrow}_0 = \emptyset \Vdash \mathbf{1} \blacktriangleright x_a = 0 \wedge x_b = 0 \wedge x_c = 0$, therefore, by the first clause, $\varphi \in S_P{\uparrow}_1$, where $\varphi = \exists \mathbf{x}'.(x'_a = 0 \wedge x'_b = 0 \wedge x'_c = 0 \wedge x_a = x'_a + 1 \wedge x_b = x'_b \wedge x_c = x'_c)$, which is equivalent to $x_a = 1 \wedge x_b = 0 \wedge x_c = 0$. From now on, we leave to the reader the details concerning equivalence of constraints. By reasoning in a similar way, using clauses 2 and 3 we calculate $S_P{\uparrow}_1$ (see Figure 8.2).

We now compute $S_P{\uparrow}_2$. By 4, as $S_P{\uparrow}_1 \Vdash a \blacktriangleright x_a = 0 \wedge x_b = 0 \wedge x_c = 0$, we get $x_a = 0 \wedge x_b = 2 \wedge x_c = 0$, and, similarly, we get $x_a \geq 0 \wedge x_b \geq 3 \wedge x_c \geq 0$. By 5, we have $x_a \geq 2 \wedge x_b \geq 0 \wedge x_c \geq 0$, while clause 6 is not (yet) applicable. Therefore, modulo redundant constraints (i.e., constraints *subsumed* by the already calculated ones) the value of $S_P{\uparrow}_2$ is given in Figure 8.2.

Now, we can compute $S_P \uparrow_3$. By 4 and $x_a \geq 2 \wedge x_b \geq 0 \wedge x_c \geq 0 \in S_P \uparrow_2$ we get $x_a \geq 1 \wedge x_b \geq 2 \wedge x_c \geq 0$, which is subsumed by $x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0$. By 5 and $x_a = 0 \wedge x_b = 2 \wedge x_c = 0$, we get $x_a = 1 \wedge x_b = 1 \wedge x_c = 0$, subsumed by $x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0$. Similarly, by 5 and $x_a \geq 0 \wedge x_b \geq 3 \wedge x_c \geq 0$ we get redundant information. By 6, from $x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0$ and $x_a = 0 \wedge x_b = 2 \wedge x_c = 0$ we get $x_a = 0 \wedge x_b = 1 \wedge x_c = 1$, from $x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0$ and $x_a \geq 0 \wedge x_b \geq 3 \wedge x_c \geq 0$ we get $x_a \geq 0 \wedge x_b \geq 2 \wedge x_c \geq 1$, and finally from $x_a \geq 2 \wedge x_b \geq 0 \wedge x_c \geq 0$ and $x_a \geq 1 \wedge x_b \geq 1 \wedge x_c \geq 0$ we have $x_a \geq 1 \wedge x_b \geq 0 \wedge x_c \geq 1$. The reader can verify that

no additional provable multisets can be obtained. It is somewhat tedious, but in no way difficult, to verify that clause 6 yields only redundant information when applied to every possible couple of constraints in $S_P{\uparrow}_3$. We have then $S_P{\uparrow}_4 = S_P{\uparrow}_3 = \mathcal{F}_{sym}(P)$, so that in this particular case we achieve termination. We can reformulate the operational semantics of P using the more suggestive multiset notation (we recall that $[\![\mathcal{A}]\!] = \{\mathcal{B} \mid \mathcal{A} \preccurlyeq \mathcal{B}\}$, where $\preccurlyeq$ is multiset inclusion):

$$F(P) = \{\{a\}, \{b,b\}, \{b,c\}\} \ \cup \ [\![\{a,b\}, \{c,c\}, \{b,b,b\}, \{a,a\}, \{b,b,c\}, \{a,c\}]\!].$$

□

Example 8.24 Let us consider the $\mathrm{LO}_{\mathbf{1}}$ program P of Example 3.13, which is given below.

1. $c \parr c \multimap \top$
2. $a \parr b \multimap \mathbf{1}$
3. $b \multimap a \otimes c$

Let $\Sigma = \{a, b, c\}$ and consider constraints over the variables $\mathbf{x} = \langle x_a, x_b, x_c \rangle$. By axioms 1 and 2 we get that

$$S_P{\uparrow}_1 = \{x_a \geq 0 \wedge x_b \geq 0 \wedge x_c \geq 2, \ x_a = 1 \wedge x_b = 1 \wedge x_c = 0\}.$$

Now we can apply clause 3 We have that $S_P{\uparrow}_1 \Vdash a \blacktriangleright x_a = 0 \wedge x_b = 1 \wedge x_c = 0$, and also $S_P{\uparrow}_1 \Vdash c \blacktriangleright x_a \geq 0 \wedge x_b \geq 0 \wedge x_c \geq 1$, therefore by the $\otimes$-case of $\Vdash$ definition we have $S_P{\uparrow}_1 \Vdash a \otimes c \blacktriangleright x_a \geq 0 \wedge x_b \geq 1 \wedge x_c \geq 1$. By definition of S_P, the constraint $x_a \geq 0 \wedge x_b \geq 2 \wedge x_c \geq 1$ belongs to $S_P{\uparrow}_2$. There are no other elements except for redundant ones, therefore we can assume

$$S_P{\uparrow}_2 = S_P{\uparrow}_1 \ \cup \{x_a \geq 0 \wedge x_b \geq 2 \wedge x_c \geq 1\}.$$

Now, we can still apply clause 3. We have that $S_P{\uparrow}_2 \Vdash c \blacktriangleright x_a \geq 0 \wedge x_b \geq 2 \wedge x_c \geq 0$. By combining this with $S_P{\uparrow}_2 \Vdash a \blacktriangleright x_a = 0 \wedge x_b = 1 \wedge x_c = 0$, we get the following judgment: $S_P{\uparrow}_2 \Vdash a \otimes c \blacktriangleright x_a \geq 0 \wedge x_b \geq 3 \wedge x_c \geq 0$. The new element is therefore $x_a \geq 0 \wedge x_b \geq 4 \wedge x_c \geq 0$, and we can assume

$$S_P{\uparrow}_3 = S_P{\uparrow}_2 \ \cup \{x_a \geq 0 \wedge x_b \geq 4 \wedge x_c \geq 0\}.$$

As the reader can verify, $S_P{\uparrow}_4 = S_P{\uparrow}_3 = \mathcal{F}_{sym}(P)$. Therefore, using the multiset notation we have that

$$F(P) = \{\{a, b\}\} \ \cup \ [\![\{c,c\}, \{b,b,c\}, \{b,b,b,b\}]\!].$$

□

8.4 An Example: Readers/Writers

We conclude the chapter with an informal discussion about the connection between linear logic specifications and the theory of broadcast protocols presented in Section 4.4.2. In particular, we will show how the readers/writers example of Section 4.4.1 can be encoded in the fragment $\mathrm{LO}_{\mathbf{1}}$. The idea, explained below, is to exploit the capability of *counting* resources given by the constant $\mathbf{1}$, together with the operational semantics of the additive conjunction $\&$, to implement the *invalidation* phase of the above protocol.

The readers/writers protocol presented in Section 4.4.1 is an abstraction of consistency protocols used for maintaining coherence in distributed database systems. We have a set of identical processes which compete for a shared resource. Processes can be reading, writing, or otherwise be idle. Processes are allowed to *concurrently* read the given resource, while writing must be an exclusive operation. This protocol can be encoded by introducing a central *monitor* which serializes the accesses to the shared resource. The monitor must guarantee mutual exclusion between agents with read and write access, and between processes with write access.

The linear logic encoding of the above protocol is as follows. First of all, let us discuss the initialization phase. As usual, we introduce a propositional symbol *init* to load the initial configuration, and the following LO clauses:

1. $init \circ\!\!- init \parr idle$
2. $init \circ\!\!- monitor$

The first clause creates an arbitrary number of *idle* processes, which can compete for the given shared resource. The second clause terminates the initialization phase by creating the central monitor. Note that the number of initial processes is a *parameter* of the protocol.

Idle processes can non-deterministically request to upgrade their rights by sending a request to the monitor. In response to such a request, the monitor (whose task is to serialize the requests) enters a state in which it starts sending invalidation messages to the other agents. The encoding is as follows.

3. $idle \parr monitor \circ\!\!- read\_req \parr read\_inv$
4. $idle \parr monitor \circ\!\!- write\_req \parr write\_inv$

Clause 3 and 4 deal with the case of *idle* processes willing to get, respectively, read access or write access. In response to the corresponding requests, named *read_req* and *write_req*,

the monitor enters an invalidation state called *read_inv* or *write_inv*. Note that after a given request has been received by the monitor, no other requests can be accepted until the first one has been processed to completion. This is enforced by requiring processes to explicitly synchronize with the atom *monitor*, and removing it from the current state after backchaining on the above clauses.

In the following we use the propositional symbols *reading* and *writing* to denote processes in the corresponding state. A simple refinement of the above specification is given by the following clause.

5. $reading \parr idle \circ\!\!- write\_req \parr write\_inv$

Namely, we allow a *reading* process to ask the monitor for an upgrade to state *writing*. As for any write request, the monitor must enter a state of write invalidation.

Let us now discuss the core of the protocol, i.e., the invalidation phase. The coherence protocol works as follows. Every time a process requests a write access, a broadcast message must be sent to all reading and writing processes, requiring they downgrade their state to *idle*. The management of read requests is similar, with the difference that only writing processes must be invalidated. The broadcast can be simulated using an invalidation cycle, as follows.

6. $read\_inv \parr writing \circ\!\!- read\_inv \parr idle$
7. $write\_inv \parr writing \circ\!\!- write\_inv \parr idle$
8. $write\_inv \parr reading \circ\!\!- write\_inv \parr idle$

The following clauses can be used to test that all processes which are required to be invalidated have moved to state *idle*. Note the use of the & connective.

9. $read\_inv \parr read\_req \circ\!\!- test\_read \,\&\, (monitor \parr reading)$
10. $write\_inv \parr write\_req \circ\!\!- test\_write \,\&\, (monitor \parr writing)$

Let us discuss for instance clause 9. Using the connective & , we split the computation into two branches. In the first one, we use the atom *test_read* to test that all writing processes have been invalidated. If this check is successful, in the other branch we grant read access to the requesting process, re-initializing the monitor, which will be ready to accept new requests afterwards. Clause 10 is similar. The test phase is implemented by the following clauses. Note the use of the constant **1** to *count* resources.

11. $test\_read \parr idle \circ\!\!- test\_read$
12. $test\_write \parr idle \circ\!\!- test\_write$
13. $test\_read \parr reading \circ\!\!- test\_read$
14. $test\_read \circ\!\!- \mathbf{1}$
15. $test\_write \circ\!\!- \mathbf{1}$

Intuitively, the test succeeds only when the resources at the moment of its first invocation do not contain occurrences of atoms representing a forbidden access (i.e., *reading* or *writing*

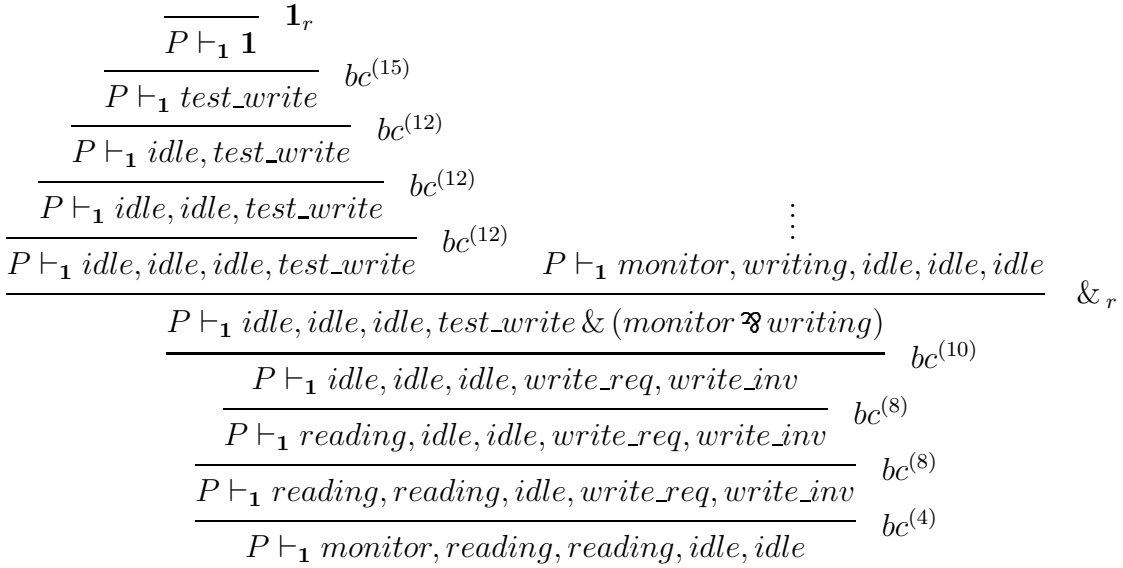

Figure 8.3: Invalidation phase for the readers/writers example

atoms for a write request and *writing* atoms for a read request). This check on the *global* state is performed using two important aspects of $\mathrm{LO}_{\mathbf{1}}$: the use of the additive conjunction $\&$, which allows to create an auxiliary branch of the current computation in which the global context is copied to (clauses 9 and 10), and the capability of counting resources introduced with the constant $\mathbf{1}$ (clauses 14 and 15).

Example 8.25 Let P the $\mathrm{LO}_{\mathbf{1}}$ program consisting of clauses 1 through 15 above. Figure 8.3 presents an LO proof showing how the invalidation phase works. We have a configuration consisting of the central monitor, two reading processes and two idle processes. One of the two idle processes asks the monitor for being granted a write access. The central monitor accepts the request and starts the invalidation phase. As a result, both reading processes are downgraded to state idle. Once invalidation has been performed, a test is made to ensure that all processes have been invalidated. The test is performed in the left branch of the proof in Figure 8.3: idle processes are removed one by one and a check is made to ensure that there are no other processes left. The right branch of the proof in Figure 8.3 represents the main branch of the computation: the requesting process is finally granted the write access and the monitor is re-initialized. As a result, the final configuration consists of the central monitor, one writing process and three idle processes. □

8.4.1 Verification of the Readers/Writers Protocol

The readers/writers protocol presented in Section 8.4 must satisfy a *mutual exclusion* property. Specifically, we wish to ensure that no read and write accesses are granted simultaneously, and that write operations are exclusive. In this section we show how the constraint bottom-up semantics for $\mathrm{LO}_\mathbf{1}$ discussed in Section 8.2 can be exploited to formally validate the readers/writers protocol with respect to the mutual exclusion property. As $\mathrm{LO}_\mathbf{1}$ specifications are currently not supported by our automatic verification tool (see Appendix A), we have evaluated the semantics for the above example by hand.

We fix the following conventions. Let Σ be the signature comprising the propositional symbols used in the specification of the readers/writers protocol of Section 8.4. In order to manage *occurrence constraints*, we associate to every symbol $a \in \Sigma$ a variable x_a (e.g. we have x_{idle}, $x_{monitor}$, and so on). Let $\mathcal{V}$ the resulting (finite) set of variables. We introduce the following compact notation. Given a constraint φ, we denote by $\varphi \wedge \mathrm{REST} \diamond 0$, where $\diamond \in \{=, \geq\}$, the constraint $\varphi \wedge x_1 \diamond 0 \wedge \ldots \wedge x_k \diamond 0$, where $\{x_1, \ldots, x_k\} = \mathcal{V} \backslash FV(\varphi)$.

First of all, we must specify the set of unsafe states of the readers/writers protocol. As usual, we introduce LO axioms to specify the *minimality violations* of the mutual exclusion property, as follows.

16. $reading \parr writing \circ\!\!- \top$
17. $writing \parr writing \circ\!\!- \top$

A configuration is unsafe if there are *at least* two processes whose access rights are in conflict.

We are now ready to compute the constraint semantics for the resulting specification, i.e., the $\mathrm{LO}_\mathbf{1}$ program consisting of clauses 1 through 17. Starting from the axioms (i.e., clauses 14 through 17), we compute the interpretation made up of the following constraints:

φ_1. $x_{test\_read} = 1 \wedge \mathrm{REST} = 0$
φ_2. $x_{test\_write} = 1 \wedge \mathrm{REST} = 0$
φ_3. $x_{reading} \geq 1 \wedge x_{writing} \geq 1 \wedge \mathrm{REST} \geq 0$
φ_4. $x_{writing} \geq 2 \wedge \mathrm{REST} \geq 0$

The remaining program clauses can now be applied to the above constraint interpretation. However, we can immediately notice that the fixpoint computation is not terminating. In fact, consider for instance clause 11, i.e., $test\_read \parr idle \circ\!\!- test\_read$. By repeatedly applying this clause to the constraint $x_{test\_read} = 1 \wedge \mathrm{REST} = 0$, we get the new constraints $x_{test\_read} = 1 \wedge x_{idle} = 1 \wedge \mathrm{REST} = 0$, $x_{test\_read} = 1 \wedge x_{idle} = 2 \wedge \mathrm{REST} = 0$, and so on. In other words, clause 11 can introduce an *arbitrary* number of *idle* atoms into any configuration containing a *test_read* atom. Similarly, clause 12 can introduce *idle* atoms into a configuration containing a *test_write* atom, and, finally, clause 13 can introduce *reading* atoms into a configuration containing a *test_read* atom.

$$\varphi'_1.\ \ x_{test\_read} = 1 \wedge x_{reading} \geq 0 \wedge x_{idle} \geq 0 \wedge \text{REST} = 0$$
$$\varphi'_2.\ \ x_{test\_write} = 1 \wedge x_{idle} \geq 0 \wedge \text{REST} = 0$$
$$\varphi_3.\ \ x_{reading} \geq 1 \wedge x_{writing} \geq 1 \wedge \text{REST} \geq 0$$
$$\varphi_4.\ \ x_{writing} \geq 2 \wedge \text{REST} \geq 0$$

Figure 8.4: Fixpoint computed using invariant strengthening for the readers/writers protocol

A way to avoid non-termination of the fixpoint computation is to exploit the methodology of *invariant strengthening* described in Section 6.1.2.1. To this aim, consider the following constraints:

$$\varphi'_1.\ \ x_{test\_read} = 1 \wedge x_{reading} \geq 0 \wedge x_{idle} \geq 0 \wedge \text{REST} = 0$$
$$\varphi'_2.\ \ x_{test\_write} = 1 \wedge x_{idle} \geq 0 \wedge \text{REST} = 0$$

These constraints formalize the intuitive idea explained above. For instance, the first one states that any configuration with one *test_read* atom and any number of *reading* and *idle* atoms (and nothing else) is reachable. Similarly for the second constraint. In other words, we have performed a *widening* operation by transforming a chain of constraints like $x = 0$, $x = 1$, $x = 2$, and so on, into a constraint like $x \geq 0$.

Consider the interpretation shown in Figure 8.4. This interpretation strictly includes the interpretation computed at the first step (i.e., without invariant strengthening). By Proposition 6.15, we are allowed to start the evaluation of the bottom-up semantics from the set of constraints in Figure 8.4. If we can prove that the modified problem is *unsatisfiable* (i.e., the atom *init* is not backward reachable) then Proposition 6.15 guarantees that also the original problem is unsatisfiable. Note that the intuition we used to compute the modified constraints φ'_1 and φ'_2 does not matter at this level: in order for the methodology of invariant strengthening to be *sound*, we only need to ensure that the set of unsafe states we start the computation from is *larger* than the original one (and this is evident for the interpretation of Figure 8.4).

It turns out that the interpretation shown in Figure 8.4 is actually the fixpoint yielded by the bottom-up computation. In order to verify this, we need to ensure that nothing else can be produced by applying clauses 1 through 13. We have the following:

- clauses 1 through 8 are not applicable to constraints φ'_1 and φ'_2 (e.g. the body of clause 1 contains an atom *init*, whereas $x_{init} = 0$ in φ'_1 and φ'_2), whereas φ_3 and φ_4 are closed w.r.t. applications of clauses 1 through 8;
- clause 9 is not applicable. In fact, the semantics of & requires to find a common context in which the two conjuncts are both provable. However, this is not possible for the following reasons. The second conjunct, $monitor \parr reading$, is only satisfiable

using φ_3 and φ_4, with an output context containing at least one writing process (i.e., $x_{writing} \geq 1$). The first conjunct, *test_read*, is satisfiable with φ'_1, but with an output context containing zero writing processes (i.e., $x_{writing} = 0$); also, the first conjunct is not satisfiable using φ'_2; finally, using φ_3 and φ_4 to satisfy the first conjunct we get φ_3 and φ_4, therefore nothing new is computed;

- clause 10 is not applicable (we can use a similar argument as for clause 9; note that the second conjunct, $monitor \parr writing$, is only satisfiable with an output context such that either $x_{reading} \geq 1$ or $x_{writing} \geq 1$, whereas the first conjunct, *test_write* is satisfiable using φ'_2 with an output context such that $x_{reading} = 0$ and $x_{writing} = 0$);
- every constraint in Figure 8.4 is clearly closed with respect to applications of clauses 11 through 13.

We can conclude that the mutual exclusion property holds for the readers/writers protocol presented in Section 8.4, independently of the number of initial processes.

8.5 Related Work

In this section we have extended the bottom-up semantics for propositional LO given in Chapter 7, by considering an extension admitting the constant **1** (and the multiplicative conjunction) in goals. Evaluating the resulting semantics amounts to computing the *reachability* set for Petri nets, and is therefore undecidable [EN94].

The model of parameterized broadcast protocols has been introduced in [EN98], where a technique extending the Karp and Miller's *coverability graph* construction for Petri nets [KM69] is proposed, and used to verify some safety properties for an invalidation-based cache coherence problem. However, as shown in [EFM99], the procedure proposed in [EN98] may not terminate (even for protocols with only broadcast moves), whereas the model-checking problem for *safety* properties for broadcast protocols is decidable. In particular, in [EFM99] it is shown that a *backward* verification procedure, along the lines of [ACJT96], can be used to check safety properties for broadcast protocols with termination guarantee. A similar result, given in [FS01], shows that the *covering* problem for Petri nets with *transfer arcs* is decidable. In [EFM99], it is also proved that the model-checking problem for *liveness* properties for broadcast protocols is undecidable. Finally, we mention that in [DEP99] efficient data structures for implementing constraint systems for broadcast protocols are studied.

Concerning our work, and in particular the broadcast protocol encoding informally presented in Section 8.4, we note that in general our verification algorithm in *not* guaranteed

to terminate (as stated in Proposition 8.2. Clearly, the constant **1** can be used to implement more powerful computational mechanisms than transfer arcs. However, as informally shown in Section 8.4.1, termination can be enforced by using suitable *acceleration* operators (see e.g. [JN00, PS00]). We will address this point as part of our future work.

Finally, we mention that in [AP91a] a different computational mechanism for LO programs, involving *broadcast communication*, is considered. This mechanism operates at the *meta-level* and is an alternative to the classical interpretation of logic programs involving the concepts of *shared logic variable*, *unification* and *computed answer substitution*. The broadcast mechanism operates as follows. Every branch of a proof tree is seen as a different *object*. Proof construction can be seen as a bidirectional process, which, starting from a partially defined initial node, either extends a branch of the proof tree, or furtherly specifies the initial node. Partially specified nodes can be instantiated every time the backchaining rule (*method activation*) is fired, and the relevant bindings are propagated *backwards* to the root of the proof tree, and, consequently, to the other branches.

Summary of the Chapter. *In this chapter we have discussed a fragment of propositional linear logic consisting of the language LO presented in the previous chapter enriched with the multiplicative conjunction and the constant* **1**. *We have extended the evaluation algorithm for computing the bottom-up semantics to this fragment. The semantics is based on a class of meta-constraints which symbolically represent provable multisets by counting resource occurrences. Though evaluation is in general non-terminating, the semantics is still effective and complete with respect to the operational semantics. We have shown that the greater expressive power of this logic can be used to simulate broadcast primitives.*

In the next chapter we will extend the language LO in another direction, namely we will discuss a first-order formulation of LO with constraints. Constraints are an elegant and convenient way to enrich logic languages with capabilities for reasoning on specialized domains. We will isolate interesting fragments for which evaluation of the bottom-up semantics is guaranteed to terminate.

Chapter 9

Reasoning on Specialized Domains: LO with Constraints

In this chapter we will extend the semantics presented in Chapter 7 by considering a first-order formulation of LO. In particular, we will consider program clauses enriched with *constraints*, as a means to reason on specialized domains. As in traditional constraint programming [JM94], reasoning on heterogeneous domains (e.g. integer or real numbers, strings and so on) is made easier by keeping the core specification logic separate from the logic dealing with the particular domain under consideration. This solution is typically more flexible because reasoning on different domains can be delegated to specialized constraint solvers.

In this chapter we will extend the construction of Chapter 7, which relates bottom-up evaluation for LO programs with verification of concurrent systems specified as Petri nets. In particular, using the concept of *constraint* we will be able to address the connection between linear logic and coloured Petri nets (see Section 5.3). Our construction will provide an *assertional language* (in the sense of [KMM$^+$97b]) to symbolically represent infinite collections of states for systems parametric in *several dimensions*. The notion of *constraint* is central to this construction. In fact, our approach is based on a combination of constraints and multiset rewriting (provided by the underlying fragment of linear logic). On the one hand, multiset rewriting allows us to *locally* specify the behaviour of most concurrent systems in a natural way. On the other hand, annotating multiset rewrite rules with constraints allows us to *finitely* and *concisely* (e.g. without the need of explicit axioms for arithmetic operations) represent transition relations. Building upon these ideas, we will introduce the notion of *constrained multiset*, which can be seen as a symbolic representation for *upward-closed* sets of markings of coloured Petri nets. Specifically, constraints will be the technical device used to represent *data* attached to processes. Similarly to Chapter 7, our construction will make use of a fixpoint operator to symbolically compute

the *predecessor* operator on interpretations consisting of sets of constrained multisets. The resulting bottom-up evaluation algorithm, according to the connection between provability and reachability (see Section 6.3), can be viewed as an alternative to the so-called *occurrence-graph* construction for coloured Petri nets [Jen97].

Technically, as in Chapter 7, we will define a bottom-up procedure to compute all goal formulas which are provable from a given program. Thanks to the use of constraint solvers, this semantics will be effective and complete with respect to the operational semantics. Furthermore, we will isolate a fragment of first-order LO, enriched with constraints, for which evaluation of the bottom-up semantics is guaranteed to terminate. We will exploit this result to prove mutual exclusion for a parameterized formulation of the so-called *ticket* protocol. As usual, for the sake of simplicity we re-use some notations like, e.g., the ones for fixpoint operators and judgments.

Some preliminary results concerning the contents of this chapter appeared in [BDM01a, BD02].

9.1 Enriching LO With Constraints

In this section we define the language LO enriched with constraints. First of all, we need to define the notion of *constraint system*.

9.1.1 Constraint Systems

Definition 9.1 (Constraint System) *A constraint system is a tuple* $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ *where:*

i. Σ *is a signature with predicates;*

ii. $\mathcal{V}$ *is a denumerable set of variables;*

iii. $\mathcal{L}$ *is a first-order language over* Σ *and* $\mathcal{V}$ (the **assertional language**) *defining a set of formulas* (the **constraints**)*, closed with respect to variable renaming, existential quantification and conjunction, and allowing equalities between variables;*

iv. $\mathcal{D}$ *is a possibly infinite set* (the **interpretation domain**)*;*

v. $Sol(\varphi)$ *is a set of mappings* $\mathcal{V} \to \mathcal{D}$ (the set of **solutions** *of a constraint* $\varphi \in \mathcal{L}$*) that preserves the usual semantics of equalities,* $\wedge$ *and* $\exists$ *(intersection and projection of the solutions);*

vi. $\sqsubseteq^c$ *is a relation such that* $\varphi \sqsubseteq^c \psi$ *implies* $Sol(\varphi) \subseteq Sol(\psi)$ (the **entailment** relation: we say that φ *entails* ψ).

We assume that $\mathcal{L}$ *contains constraints, denoted* $true$ *and* $false$*, which are identically true and identically false in* $\mathcal{D}$*.*

By analogy with constraint programming, further requirements on constraint systems, like *solution compactness* [JL87], can be imposed. We refer to [JL87, Mah92] for a discussion.

In the following we will refer to a generic mapping $\mathcal{V} \to \mathcal{D}$ as an **evaluation** for the variables in $\mathcal{V}$ into $\mathcal{D}$. We use the notation $\langle x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots \rangle$ to denote an evaluation mapping x_1 to d_1, x_2 to d_2, and so on, and the notation $\sigma_{|\mathbf{x}}$ to denote the *restriction* of the evaluation σ to the variables $\mathbf{x}$. We also say that a constraint φ is **satisfiable** if $Sol(\varphi) \neq \emptyset$.

We conclude this section showing some examples of constraint systems which we will need later on in this chapter.

Definition 9.2 (The Herbrand Constraint System HC) *Let* Σ *be a signature and* $\mathcal{V}$ *a denumerable set of variables. We define the Herbrand constraint system* $\mathrm{HC} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$*, where:* $\mathcal{D}$ *is the set of non-ground terms in* $T^{\mathcal{V}}_{\Sigma}$*,* $\mathcal{L}$ *contains equalities between non-ground terms in* $T^{\mathcal{V}}_{\Sigma}$*, equality is interpreted as unification between terms, Sol maps constraints to substitutions (term unifiers) and* $\tau \sqsubseteq^c \theta$ *if* $\theta \leq \tau$ *(i.e.,* θ *is* more general *than* τ*).*

Definition 9.3 (The Constraint System LC) *The class of* **linear integer constraints** *(*LC*-constraints) consists of integer constraints of the form*

$$\varphi \ ::= \ \varphi \wedge \varphi \mid a_1 x_1 + \ldots + a_n x_n = a_{n+1} \mid a_1 x_1 + \ldots + a_n x_n > a_{n+1} \mid true \mid false$$

where $a_i \in \mathbb{Z}$ *for* $i : 1, \ldots, n+1$*. Given* $\mathcal{D} = \mathbb{Z}$*, the interpretation Sol maps constraints into sets of variable evaluations from* $\mathcal{V}$ *to* $\mathbb{Z}$*, and* $\sqsubseteq^c$ *is the usual entailment relation between integer constraints.*

Definition 9.4 (The Constraint System DC) *The class of* **difference constraints** *(*DC*-constraints) is the subclass of linear integer constraints having the form*

$$\varphi \ ::= \ \varphi \wedge \varphi \mid x = y + c \mid x > y + c \mid true \mid false$$

where $c \in \mathbb{Z}$*, Given* $\mathcal{D} = \mathbb{Z}$*, the interpretation Sol maps constraints into sets of variable evaluations from* $\mathcal{V}$ *to* $\mathbb{Z}$*, and* $\sqsubseteq^c$ *is the usual entailment relation for linear integer constraints.*

Definition 9.5 (The Constraint System NC) *The class of* **name constraints** *(*NC*-constraints) is the subclass of difference constraints having the form*

$$\varphi \ ::= \ \varphi \wedge \varphi \mid x = y \mid x > y \mid true \mid false$$

interpreted over $\mathbb{Z}$ *and ordered with respect to the entailment relation* $\sqsubseteq^c$ *of linear integer constraints.*

Definition 9.6 (The Constraint System EC) *The class of* **equality constraints** *(*EC*-constraints) is the subclass of name constraints having the form*

$$\varphi \ ::= \ \varphi \wedge \varphi \mid x = y \mid true \mid false$$

interpreted over $\mathbb{Z}$ *and ordered with respect to the entailment relation* $\sqsubseteq^c$ *of linear integer constraints.*

For *linear integer constraints* (and thus DC-, NC- and EC-constraints) it is well-known that there are algorithms for checking satisfiability, entailment, and for variable elimination (see e.g. [BGP97]).

Example 9.7 Let φ be $x > y \wedge x > z$, then $\sigma = \langle x \mapsto 2, y \mapsto 1, z \mapsto 0, \ldots \rangle \in Sol(\varphi)$. Furthermore, φ is satisfiable, $\varphi \sqsubseteq^c (x > y)$, and $\exists y.\varphi \equiv (x > z)$. □

We are now ready to define the language LO($\mathcal{C}$).

9.1.2 The Language LO($\mathcal{C}$)

In order to define the language LO($\mathcal{C}$), we will make a simplifying assumption. Namely, we assume that LO($\mathcal{C}$) programs are built over atomic formulas consisting of a predicate symbol applied to a list of *distinct variables*, instead of arbitrary terms. This assumption simplifies a lot the following presentation, without causing any loss of generality. In fact, arbitrary atomic formulas can be recovered by considering a constraint system in which *equality* is interpreted as *unification* on a first-order term language (see Definition 9.2). We give the following definitions.

Definition 9.8 (Atomic formulas) *Let* Π *be a finite set of predicate symbols, and* $\mathcal{V}$ *a denumerable set of variables. An atomic formula over* Π *and* $\mathcal{V}$ *has the form* $p(x_1, \ldots, x_n)$ *(with* $n \geq 0$*), where* $p \in \Pi$*, and* $x_1, \ldots, x_n$ *are distinct variables in* $\mathcal{V}$*.*

Definition 9.9 (LO($\mathcal{C}$) programs) *Let Π be a finite set of predicate symbols, and $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ a constraint system. The sets of LO($\mathcal{C}$) goal formulas, head formulas, and clauses, over Π and $\mathcal{C}$, are defined by the following grammar*

$$\begin{aligned} \mathbf{G} &::= \mathbf{G} \parr \mathbf{G} \mid \mathbf{G} \,\&\, \mathbf{G} \mid \mathbf{A} \mid \top \mid \bot \\ \mathbf{H} &::= \mathbf{A} \parr \ldots \parr \mathbf{A} \mid \bot \\ \mathbf{D} &::= \forall(\mathbf{H} \circ\!\!- \mathbf{G} \,\square\, \varphi) \mid \mathbf{D} \,\&\, \mathbf{D} \end{aligned}$$

where $\mathbf{A}$ is an atomic formula over Π and $\mathcal{V}$, and φ is a satisfiable constraint in $\mathcal{L}$. We assume that all variables appearing in a $\mathbf{G}$-formula, $\mathbf{H}$-formula or $\mathbf{D}$-formula are distinct from each other. An LO($\mathcal{C}$) program is a $\mathbf{D}$-formula over Π and $\mathcal{C}$.

Constraints can be viewed as a convenient way to represent (infinite) sets of ground clauses. Therefore, a simple way to define the operational semantics of the language LO($\mathcal{C}$) is to re-formulate the notion of *ground instance* of a program given in Definition 3.7. In the rest of the chapter, we will use the notation $\sigma(F)$, where F is any expression (e.g. an LO($\mathcal{C}$) clause) with variables in $\mathcal{V}$ and σ is a mapping $\mathcal{V} \to \mathcal{D}$, to denote the application of σ to F.

Definition 9.10 (Ground Instances) *Let Π be a finite set of predicate symbols, and $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ a constraint system. Given an LO($\mathcal{C}$) program P, the set of ground instances of P, denoted $Gnd(P)$, is defined as follows:*

$$Gnd(P) = \{\sigma(H \circ\!\!- G) \mid (H \circ\!\!- G \,\square\, \varphi) \in P \text{ and } \sigma \in Sol(\varphi)\}$$

Example 9.11 Let $\Pi = \{p, q, r, s\}$, and consider unification constraints over the term language generated by a constant symbol a and a function symbol f. Let $\mathcal{V}$ be a denumerable set of variables and $x, y, \ldots \in \mathcal{V}$. Let C be the clause

$$p(x) \parr p(z) \circ\!\!- (q(x') \parr r(y)) \,\&\, s(z') \,\square\, z = a \wedge x' = x \wedge z' = a.$$

Then

$$p(f(f(a))) \parr p(a) \circ\!\!- (q(f(f(a))) \parr r(f(a))) \,\&\, s(a) \in Gnd(C).$$

□

Example 9.12 Let $\Pi = \{p, q, r, s\}$, let $\mathcal{V}$ be a denumerable set of variables and $x, y, \ldots \in \mathcal{V}$. Let C be the LO(DC) clause

$$p(x) \parr q(y) \circ\!\!- q(z) \parr r(w) \parr s(w') \,\square\, z = x \wedge w > y \wedge w' = w + 1$$

Then

$$p(1) \parr q(2) \circ\!\!- q(1) \parr r(5) \parr s(6) \in Gnd(C).$$

□

$$\frac{}{P \vdash \top, \Delta}\ \top_r \qquad \frac{P \vdash G_1, G_2, \Delta}{P \vdash G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash G_1, \Delta \quad P \vdash G_2, \Delta}{P \vdash G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash \Delta}{P \vdash \bot, \Delta}\ \bot_r \qquad \frac{P \vdash G, \mathcal{A}}{P \vdash \widehat{H}, \mathcal{A}}\ bc\ \ (H \circ\!\!- G\ \in Gnd(P))$$

Figure 9.1: A proof system for LO($\mathcal{C}$)

Using definition 9.10, the notion of provability can be defined in the same way as for standard LO. The corresponding proof system is shown in Figure 9.1 for convenience. As usual, an LO($\mathcal{C}$) sequent has the form $P \vdash G_1, \ldots, G_k$, where $P = D_1 \,\&\, \ldots \,\&\, D_n$ is an LO program (the *set* of clauses $D_1, \ldots, D_n$) and $G_1, \ldots, G_k$ is a multiset of goals. A sequent is provable if all branches of its proof tree terminate with an instance of the $\top_r$ axiom. Rule *bc* is applicable only if the right-hand side of the current sequent consists of atomic formulas.

We can formulate the following proposition, which is analogous to Proposition 3.9.

Proposition 9.13 (Admissibility of the Weakening Rule) *Given an LO($\mathcal{C}$) program P and two multisets of goals Δ, Δ' such that $\Delta \preccurlyeq \Delta'$, if $P \vdash \Delta$ then $P \vdash \Delta'$.*

9.2 An Effective Semantics for LO($\mathcal{C}$)

In this section we will discuss the definition of an effective bottom-up semantics for LO($\mathcal{C}$). First of all, we reformulate the definition of the operational semantics as follows.

Definition 9.14 (Operational Semantics) *Given an LO($\mathcal{C}$) program P, its operational semantics, denoted $O(P)$, is given by*

$$O(P) = \{\mathcal{A} \mid \mathcal{A} \text{ is a multiset of ground atomic formulas and } P \vdash \mathcal{A}\}$$

Given that LO($\mathcal{C}$) provability can be reduced to LO provability using the concept of ground instances of a clause, as explained in Section 9.1, we can re-use the results of Section 7.1 in order to define a bottom-up semantics for LO($\mathcal{C}$).

An LO($\mathcal{C}$) program can be seen as a (potentially infinite) set of ground clauses. We can therefore reformulate the bottom-up semantics of Section 7.1 with the following slight modifications: we consider a (generally infinite) set of *ground* atomic formulas instead of a finite set of propositional symbols Σ; we substitute $Gnd(P)$ (for an LO($\mathcal{C}$) program P)

in place of P in all the definitions of Section 7.1. For instance, the Herbrand base will be the set of all multisets of ground atomic formulas. With these modifications, the results of Section 7.1 still hold, the proofs being exactly analogous. Infiniteness of the Herbrand base and of the set of ground clauses of a program do not cause any harm. The resulting fixpoint semantics is therefore sound and complete with respect to the operational semantics defined above. Clearly, this bottom-up semantics is not effective.

In the rest of this section, we will refer to the definitions and notations used in Section 7.1. In particular, the (concrete) satisfiability judgment $\models$ and the T_P fixpoint operator are defined, respectively, in Definition 7.4 and Definition 7.7.

We are now ready to present the symbolic version of the bottom-up semantics for LO($\mathcal{C}$). First of all, we introduce the notion of *constrained multiset*, which is central to the semantics definition.

Definition 9.15 (Constrained Multiset) *Let Π be a finite set of predicate symbols, and $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ a constraint system. A constrained multiset over Π and $\mathcal{C}$ is a multiset of atomic formulas, annotated with a constraint, of the form*

$$\{p_1(x_{11}, \ldots, x_{1k_1}), \ldots, p_n(x_{n1}, \ldots, x_{nk_n})\} : \varphi$$

where $p_1, \ldots, p_n \in \Pi$, $\varphi \in \mathcal{L}$ is a satisfiable constraint, and $x_{11}, \ldots, x_{nk_n}$ are distinct variables in $\mathcal{V}$.

We often omit brackets in constrained multiset notation.

Notation. By analogy with multiset unifiers (see Section 2.4), we define the following notation. Given two constrained multisets $\mathcal{N} = N_1, \ldots, N_n : \psi$ and $\mathcal{M} = M_1, \ldots, M_n : \varphi$ with disjoint variables (note that $|\mathcal{N}| = |\mathcal{M}|$), where $N_i = p_i(y_{i1}, \ldots, y_{ik_i})$ and $M_i = q_i(x_{i1}, \ldots, x_{ik_i})$ for $i : 1, \ldots, n$, we define the constraint $\mathcal{N} = \mathcal{M}$ as $\bigwedge_{i:1,\ldots,n}(y_{i1} = x_{l_i 1} \wedge \ldots \wedge y_{ik_i} = x_{l_i k_i})$, provided $p_i = q_{l_i}$ for every $i : 1, \ldots, n$ and $\{l_1, \ldots, l_n\}$ is a permutation of $\{1, \ldots, n\}$. We will use the notation $\mathcal{N} = \mathcal{M}$ to denote a constraint which is *non-deterministically* picked from the set of constraints which can obtained in the above manner (in general more alternatives are possible, depending on the choice of the permutation).

Constrained multisets will be used as symbolic representations for sets of elements of the (concrete) Herbrand base. With this in mind, we give the following definitions.

Definition 9.16 (Abstract Herbrand Base) *Let Π be a finite set of predicate symbols, and $\mathcal{C}$ a constraint system. Given an LO($\mathcal{C}$) program P, the Herbrand base of P, denoted $HB(P)$, is given by*

$$HB(P) = \{\mathcal{M} : \varphi \mid \mathcal{M} : \varphi \text{ is a constrained multiset over } \Pi \text{ and } \mathcal{C}\}.$$

Definition 9.17 (Abstract Interpretations) *Let* Π *be a finite set of predicate symbols, and* $\mathcal{C}$ *a constraint system. Given an* $LO(\mathcal{C})$ *program* P*, an interpretation* I *is any subset of* $HB(P)$*, i.e.,* $I \in \mathcal{P}(HB(P))$*.*

By analogy with definition 9.10, we give the following.

Definition 9.18 (Ground Instance Operator) *Let* Π *be a finite set of predicate symbols, and* $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ *a constraint system. Given an interpretation* I*, we define the operator Inst as follows:*

$$Inst(I) = \{\sigma(\mathcal{M}) \mid \mathcal{M} : \varphi \in I,\ \sigma \in Sol(\varphi)\}.$$

Let Up be the following operator on multisets of *ground* atomic formulas: $Up(I) = \{\mathcal{A} + \mathcal{C} \mid \mathcal{A} \in I\}$. The following definition provides the connection between concrete interpretations and abstract interpretations.

Definition 9.19 (Denotation of an Interpretation) *Let* Π *be a finite set of predicate symbols, and* $\mathcal{C}$ *a constraint system. Given an (abstract) interpretation* I*, its denotation* $[\![I]\!]$ *is the (concrete) interpretation defined as follows:*

$$[\![I]\!] = Up(Inst(I)).$$

Two interpretations I *and* J *are said to be equivalent, written* $I \simeq J$*, if and only if* $[\![I]\!] = [\![J]\!]$*.*

We are now ready to define the abstract interpretation domain. As usual, we can identify interpretations having the same denotation using equivalence classes with respect to the corresponding equivalence relation $\simeq$.

Definition 9.20 (Abstract Interpretation Domain) *Abstract interpretations form a complete lattice* $\langle \mathcal{I}, \sqsubseteq \rangle$*, where*

- $\mathcal{I} = \{[I]_\simeq \mid I$ *is an interpretation*$\}$*;*
- $[I]_\simeq \sqsubseteq [J]_\simeq$ *iff* $[\![I]\!] \subseteq [\![J]\!]$*;*
- *the least upper bound of* $[I]_\simeq$ *and* $[J]_\simeq$*, written* $[I]_\simeq \bigsqcup [J]_\simeq$*, is* $[I \cup J]_\simeq$*;*
- *the bottom and top elements are* $[\emptyset]_\simeq$ *and* $[\epsilon : true]_\simeq$*, respectively.*

Before going on with the definition of the satisfiability judgment, we introduce the following notion of *entailment* between constrained multisets.

Definition 9.21 (Entailment of Constrained Multisets) *Let Π be a finite set of predicate symbols, and $\mathcal{C}$ a constraint system. We call a relation $\sqsubseteq^M$ between constrained multisets an entailment whenever $\mathcal{N} : \psi \sqsubseteq^M \mathcal{M} : \varphi$ implies $[\![\mathcal{N} : \psi]\!] \subseteq [\![\mathcal{M} : \varphi]\!]$, for any constrained multisets $\mathcal{N} : \psi$ and $\mathcal{M} : \varphi$.*

An example of entailment relation is given below. As we will prove in the following, it amounts to an effective and *sufficient* condition for testing the $\sqsubseteq$ relation over interpretations. We first give the following definition.

Definition 9.22 (Entailment $\sqsubseteq^m$) *Let Π be a finite set of predicate symbols, $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ a constraint system, and $\mathcal{N} : \psi$, $\mathcal{M} : \varphi$ two constrained multisets over Π and $\mathcal{C}$ with disjoint variables. We say that $\mathcal{N} : \psi$ entails $\mathcal{M} : \varphi$, written $\mathcal{N} : \psi \sqsubseteq^m \mathcal{M} : \varphi$, if there exists a multiset $N' \preccurlyeq \mathcal{N}$ such that*

$$\exists \mathbf{x}.(\mathcal{N}' = \mathcal{M} \wedge \psi) \sqsubseteq^c \varphi$$

where $\mathbf{x} = FV(\mathcal{N} : \psi)$.

Example 9.23 Let $\mathcal{N} : \psi \equiv p(y), q(z) : y > 0 \wedge z > 0$, and $\mathcal{M} : \varphi \equiv p(x) : x > 0$. Then $\mathcal{N} : \psi \sqsubseteq^m \mathcal{M} : \varphi$. In fact, if we take $\mathcal{N}' = p(y)$ we have that $\exists y, z.(x = y \wedge y > 0 \wedge z > 0) \sqsubseteq^c x > 0$. □

The following result states that $\sqsubseteq^m$ is an entailment.

Proposition 9.24 *Let Π be a finite set of predicate symbols, $\mathcal{C}$ a constraint system, and $\mathcal{N} : \psi$, $\mathcal{M} : \varphi$ two constrained multisets over Π and $\mathcal{C}$ with disjoint variables. If $\mathcal{N} : \psi \sqsubseteq^m \mathcal{M} : \varphi$ then $[\![\mathcal{N} : \psi]\!] \subseteq [\![\mathcal{M} : \varphi]\!]$.*

Proof It is sufficient to prove that for every $\sigma \in Sol(\psi)$ there exists $\sigma' \in Sol(\varphi)$ such that $\sigma'(\mathcal{M}) \preccurlyeq \sigma(\mathcal{N})$. Let $\sigma \in Sol(\psi)$.

By hypothesis, there exists a multiset $\mathcal{N}' \preccurlyeq \mathcal{N}$ s.t. $\mathbf{x} = FV(\mathcal{N} : \psi)$ and $\exists \mathbf{x}.(\mathcal{N}' = \mathcal{M} \wedge \psi) \sqsubseteq^c \varphi$, i.e., $Sol(\exists \mathbf{x}.(\mathcal{N}' = \mathcal{M} \wedge \psi)) \subseteq Sol(\varphi)$.

Now, let $\mathcal{M} = \{M_1, \ldots, M_n\}$, $\mathcal{N}' = N_1, \ldots, N_n$ and $\{i_1, \ldots, i_n\}$ the permutation of $\{1, \ldots, n\}$ (satisfying the above relation) such that $N_{i_k} = M_k$ for $k : 1, \ldots, n$. Assuming $N_{i_k} = p(\mathbf{y}_{i_k})$ and $M_k = p(\mathbf{x}_k)$, where $\mathbf{y}_{i_k}$ and $\mathbf{x}_k$ are *vector* of variables and $p \in \Pi$, the above constraint is equivalent to $\mathbf{y}_{i_k} = \mathbf{x}_k$.

Now, let σ' be the evaluation such that ${\sigma'}_{|\mathbf{x}_k} = \sigma_{|\mathbf{y}_{i_k}}$ for every $k : 1, \ldots, n$, and $\sigma' = \sigma$ otherwise. It easy to see that $\sigma' \in Sol(\exists \mathbf{x}.(\mathcal{N}' = \mathcal{M} \wedge \psi))$ (note that the constraint $\mathcal{N}' = \mathcal{M}$ is simply a *renaming* of variables: variables inside constrained multisets are distinct and $\mathcal{N} : \psi$, $\mathcal{M} : \varphi$ have disjoint variables). Therefore $\sigma' \in Sol(\varphi)$ using the fact that $Sol(\exists \mathbf{x}.(\mathcal{N}' = \mathcal{M} \wedge \psi)) \subseteq Sol(\varphi)$. Besides, $\sigma'(\mathcal{M}) = \sigma(\mathcal{N}')$ (by definition of σ') and $\sigma(\mathcal{N}') \preccurlyeq \sigma(\mathcal{N})$, from which the conclusion. □

We extend the entailment relation to interpretations as follows.

Definition 9.25 *Let I and J be two interpretations. We say that I entails J, written $I \sqsubseteq^i J$, if for every $(\mathcal{N} : \psi) \in I$ there exists $(\mathcal{M} : \varphi) \in J$ s.t. $\mathcal{N} : \psi \sqsubseteq^m \mathcal{M} : \varphi$.*

Corollary 9.26 *Given two interpretations I and J, if $I \sqsubseteq^i J$ then $[\![I]\!] \subseteq [\![J]\!]$.*

Proof By Proposition 9.24, observing that $[\![I]\!] = \{[\![\mathcal{M} : \varphi]\!] \mid \mathcal{M} : \varphi \in I\}$. □

Remark 9.27 We note that the reverse implication in Proposition 9.24 does not hold. In fact, consider the following counterexample over linear integer constraints. Let

$$\mathcal{N} : \psi \equiv p(x, y), p(w, z) : even(x) \wedge odd(z) \wedge y = w$$
$$\mathcal{M} : \varphi \equiv p(x', y') : even(x') \wedge odd(y')$$

where $even(v)$ is a shorthand for $\exists u.(v = 2u)$ and $odd(v)$ is a shorthand for $\exists u.(v = 2u+1)$. Then it is easy to see that $[\![N : \psi]\!] \subseteq [\![M : \varphi]\!]$, while it is not the case that $N : \psi \sqsubseteq^m M : \varphi$. We also note that the reverse implication in Corollary 9.26 does not hold for a further reason (besides the one above). Consider in fact the following counterexample (over linear integer constraints): $I = \{p(x) : 1 < x < 5\}$ and $J = \{p(y) : 1 < y < 3,\ p(z) : 2 < z < 6\}$.

We are now ready to define the abstract satisfiability judgment. The judgment has the form $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$, where I is an interpretation, $\Delta : \varphi$ is a *constrained* context (i.e., a *constrained* multiset of goals) with φ *satisfiable*, $\mathcal{C}$ is an output fact (i.e., a multiset of atomic formulas), and φ' is an output constraint.

Definition 9.28 (Abstract Satisfiability Judgment) *Let P be an $LO(\mathcal{C})$ program and I an interpretation. The abstract satisfiability judgment $\Vdash$ is defined as follows:*

$I \Vdash \top, \Delta : \varphi \blacktriangleright \epsilon \blacktriangleright \varphi$;

$I \Vdash \mathcal{A} : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ *if there exist* $\mathcal{B} : \psi \in I$ *(variant),* $\mathcal{B}' \preccurlyeq \mathcal{B}$, $\mathcal{A}' \preccurlyeq \mathcal{A}$, $|\mathcal{B}'| = |\mathcal{A}'|$, $\mathcal{C} = \mathcal{B} \backslash \mathcal{B}'$, *and* $\varphi' \equiv \mathcal{B}' = \mathcal{A}' \wedge \varphi \wedge \psi$ *is satisfiable*;

$I \Vdash G_1 \,\&\, G_2, \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ *if* $I \Vdash G_1, \Delta : \varphi \blacktriangleright \mathcal{C}_1 \blacktriangleright \varphi_1$, $I \Vdash G_2, \Delta : \varphi \blacktriangleright \mathcal{C}_2 \blacktriangleright \varphi_2$, $\mathcal{D}_1 \preccurlyeq \mathcal{C}_1$, $\mathcal{D}_2 \preccurlyeq \mathcal{C}_2$, $|\mathcal{D}_1| = |\mathcal{D}_2|$, $\mathcal{C} = \mathcal{C}_1 + (\mathcal{C}_2 \backslash \mathcal{D}_2)$, *and* $\varphi' \equiv \mathcal{D}_1 = \mathcal{D}_2 \wedge \varphi_1 \wedge \varphi_2$ *is satisfiable*;

$I \Vdash G_1 \parr G_2, \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ *if* $I \Vdash G_1, G_2, \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$;

$I \Vdash \bot, \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ *if* $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$.

Example 9.29 Let $\Pi = \{p, q, r, s, t\}$, and consider unification constraints over the term language generated by a constant symbol a and a function symbol f. Let $\mathcal{V}$ be a denumerable set of variables and $x, y, z, \ldots \in \mathcal{V}$. Consider the following clause

$$p(x) \parr p(z) \circ\!\!- (q(x') \parr r(y)) \,\&\, s(z') \;\square\; z = a \wedge x' = x \wedge z' = a.$$

and the interpretation I consisting of the following constrained multisets:

$$\mathcal{M}_1 : \psi_1 \equiv t(u_1, v_1), q(w_1) : u_1 = f(q_1) \wedge w_1 = a$$
$$\mathcal{M}_2 : \psi_2 \equiv t(u_2, v_2), s(w_2) : v_2 = f(q_2) \wedge w_2 = a$$

Let $G : \varphi$ be $(q(x') \parr r(y)) \,\&\, s(z') : z = a \wedge x' = x \wedge z' = a$. First of all, we use the $\&$-rule for $\Vdash$. We have to compute $\mathcal{C}_1$, $\mathcal{C}_2$, φ_1 and φ_2 such that

$$I \Vdash q(x') \parr r(y) : \varphi \blacktriangleright \mathcal{C}_1 \blacktriangleright \varphi_1 \quad \text{and} \quad I \Vdash s(z') : \varphi \blacktriangleright \mathcal{C}_2 \blacktriangleright \varphi_2.$$

For the first conjunct and using the $\parr$-rule, we have that $I \Vdash q(x') \parr r(y) : \varphi \blacktriangleright \mathcal{C}_1 \blacktriangleright \varphi_1$ iff $I \Vdash q(x'), r(y) : \varphi \blacktriangleright \mathcal{C}_1 \blacktriangleright \varphi_1$. By the second rule for $\Vdash$, applied to $\mathcal{M}_1 : \psi_1 \in I$, we have that $\mathcal{C}_1 = t(u_1, v_1)$ and $\varphi_1 \equiv x' = w_1 \wedge w_1 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge u_1 = f(q_1)$. For the second conjunct and using the second rule for $\Vdash$ applied to $\mathcal{M}_2 : \psi_2 \in I$, we have that $\mathcal{C}_2 = t(u_2, v_2)$ and $\varphi_2 \equiv z' = w_2 \wedge w_2 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge v_2 = f(q_2)$. Therefore by definition of the $\&$-rule, if we unify $t(u_1, v_1)$ and $t(u_2, v_2)$, we have that

$$I \Vdash G : \varphi \blacktriangleright t(u_1, v_1) \blacktriangleright u_1 = u_2 \wedge v_1 = v_2 \wedge \varphi_1 \wedge \varphi_2.$$

We also have, again by the $\&$-rule (by choosing empty sub-multisets), that

$$I \Vdash G : \varphi \blacktriangleright t(u_1, v_1), t(u_2, v_2) \blacktriangleright \varphi_1 \wedge \varphi_2.$$

□

The following lemma states a simple property which we will need later.

Lemma 9.30 *For every interpretation I, constrained multiset of goals $\Delta : \varphi$, fact $\mathcal{C}$, and constraint φ', if $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ then there exists a constraint ψ s.t. $\varphi' \equiv \varphi \wedge \psi$, φ' is satisfiable, $FV(\psi) \cap FV(\varphi) \subseteq FV(\Delta)$, $FV(\mathcal{C}) \cap FV(\varphi) = \emptyset$, and $FV(\mathcal{C}) \cap FV(\Delta) = \emptyset$.*

Proof By simple induction on the $\Vdash$ definition. □

The connection between the satisfiability judgments $\models$ and $\Vdash$ is clarified by the following lemma.

Lemma 9.31 *For every interpretation I, constrained multiset of goals $\Delta : \varphi$, facts $\mathcal{C}$ and $\mathcal{S}$, constraint φ' and evaluation σ,*

i. if $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ and $\sigma \in Sol(\varphi')$, then $[\![I]\!] \models \sigma(\Delta) \blacktriangleright \mathcal{S}$ for all $\mathcal{S}$ s.t $\sigma(\mathcal{C}) \preccurlyeq \mathcal{S}$;

ii. if $[\![I]\!] \models \sigma(\Delta) \blacktriangleright \mathcal{S}$ and $\sigma \in Sol(\varphi)$, then there exist a multiset $\mathcal{C}$, a satisfiable constraint φ', and an evaluation $\sigma' \in Sol(\varphi')$ s.t. $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$, $\sigma'(\mathcal{C}) \preccurlyeq \mathcal{S}$, and $\sigma'_{|FV(\Delta)} = \sigma_{|FV(\Delta)}$.

Proof

i. By induction on the derivation of $I \Vdash \Delta : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$.

- If $I \Vdash \top, \Delta : \varphi \blacktriangleright \epsilon \blacktriangleright \varphi$ and $\sigma \in Sol(\varphi)$ then $[\![I]\!] \models \sigma(\top, \Delta) \blacktriangleright \mathcal{S}$ for every $\mathcal{S}$;
- if $I \Vdash \mathcal{A} : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ and $\sigma \in Sol(\varphi')$, then there exists a variant $\mathcal{B} : \psi$ of an element in I s.t. $\mathcal{B}' \preccurlyeq \mathcal{B}$, $\mathcal{A}' \preccurlyeq \mathcal{A}$, $|\mathcal{B}'| = |\mathcal{A}'|$, $\varphi' \equiv \mathcal{B}' = \mathcal{A}' \wedge \varphi \wedge \psi$ is satisfiable, and $\mathcal{C} = \mathcal{B} \backslash \mathcal{B}'$.

  We have that $[\![I]\!] \models \sigma(\mathcal{A}) \blacktriangleright \mathcal{S}$ if $\sigma(\mathcal{A}) + \mathcal{S} \in [\![I]\!]$. This holds if $\sigma(\mathcal{B}) \preccurlyeq \sigma(\mathcal{A}) + \mathcal{S}$ iff (remember that $\mathcal{B}' \preccurlyeq \mathcal{B}$ and $\mathcal{A}' \preccurlyeq \mathcal{A}$) $\sigma(\mathcal{B} \backslash \mathcal{B}') + \sigma(\mathcal{B}') \preccurlyeq \sigma(\mathcal{A} \backslash \mathcal{A}') + \sigma(\mathcal{A}') + \mathcal{S}$ iff $\sigma(\mathcal{B} \backslash \mathcal{B}') \preccurlyeq \sigma(\mathcal{A} \backslash \mathcal{A}') + \mathcal{S}$ iff $\sigma(\mathcal{C}) \preccurlyeq \sigma(\mathcal{A} \backslash \mathcal{A}') + \mathcal{S}$. The latter relation holds if $\sigma(\mathcal{C}) \preccurlyeq \mathcal{S}$;
- if $I \Vdash G_1 \,\&\, G_2, \Delta : \varphi \blacktriangleright \mathcal{A} \blacktriangleright \varphi'$ and $\sigma \in Sol(\varphi')$, then

  $$I \Vdash G_1, \Delta : \varphi \blacktriangleright \mathcal{A}_1 \blacktriangleright \varphi_1 \quad \text{and} \quad I \Vdash G_2, \Delta : \varphi \blacktriangleright \mathcal{A}_2 \blacktriangleright \varphi_2,$$

  with $\mathcal{A}'_1 \preccurlyeq \mathcal{A}_1$, $\mathcal{A}'_2 \preccurlyeq \mathcal{A}_2$, $|\mathcal{A}'_1| = |\mathcal{A}'_2|$, $\varphi' \equiv \mathcal{A}'_1 = \mathcal{A}'_2 \wedge \varphi_1 \wedge \varphi_2$ is satisfiable, and $\mathcal{A} = \mathcal{A}_1 + (\mathcal{A}_2 \backslash \mathcal{A}'_2)$.

  Clearly, $\sigma \in Sol(\varphi')$ implies $\sigma \in Sol(\varphi_1)$ and $\sigma \in Sol(\varphi_2)$. Therefore, by inductive hypothesis, we have that $[\![I]\!] \models \sigma(G_1, \Delta) \blacktriangleright \mathcal{S}$ for all $\mathcal{S}$ s.t. $\sigma(\mathcal{A}_1) \preccurlyeq \mathcal{S}$, and $[\![I]\!] \models \sigma(G_2, \Delta) \blacktriangleright \mathcal{S}$ for all $\mathcal{S}$ s.t. $\sigma(\mathcal{A}_2) \preccurlyeq \mathcal{S}$.

  Clearly, $\sigma(G_1, \Delta) = \sigma(G_1), \sigma(\Delta)$, and similarly for G_2. Therefore we have that $[\![I]\!] \models \sigma(G_1) \,\&\, \sigma(G_2), \sigma(\Delta) \blacktriangleright \mathcal{S}$ for all $\mathcal{S}$ s.t. $\sigma(\mathcal{A}_1) \preccurlyeq \mathcal{S}$ and $\sigma(\mathcal{A}_2) \preccurlyeq \mathcal{S}$, i.e., $[\![I]\!] \models \sigma(G_1 \,\&\, G_2, \Delta) \blacktriangleright \mathcal{S}$ for all $\mathcal{S}$ s.t. $\sigma(\mathcal{A}_1) \preccurlyeq \mathcal{S}$ and $\sigma(\mathcal{A}_2) \preccurlyeq \mathcal{S}$.

  In order to conclude, it is sufficient to observe that $\sigma(\mathcal{A}_1) \preccurlyeq \sigma(\mathcal{A})$ and $\sigma(\mathcal{A}_2) \preccurlyeq \sigma(\mathcal{A})$. In fact, $\mathcal{A} = \mathcal{A}_1 + (\mathcal{A}_2 \backslash \mathcal{A}'_2)$, so that $\sigma(\mathcal{A}_1) \preccurlyeq \sigma(\mathcal{A})$. Besides, $\sigma(\mathcal{A}_2) \preccurlyeq \sigma(\mathcal{A})$ because $\sigma(\mathcal{A}'_2) = \sigma(\mathcal{A}'_1)$ and $\sigma(\mathcal{A}'_1) \preccurlyeq \sigma(\mathcal{A}_1)$;
- if $\Delta = G_1 \parr G_2, \Delta'$ of $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[\![I]\!] \models \sigma(\Delta) \blacktriangleright \mathcal{S}$.

- If $[\![I]\!] \models \top, \Delta \blacktriangleright \mathcal{S}$, $\sigma \in Sol(\varphi)$ and $\Delta = \sigma(\Delta_1)$, then we take $\mathcal{C} = \epsilon$ and $\sigma' = \sigma$, and we have $I \Vdash \top, \Delta_1 : \varphi \blacktriangleright \epsilon \blacktriangleright \varphi$;
- if $[\![I]\!] \models \mathcal{A} \blacktriangleright \mathcal{S}$, $\sigma \in Sol(\varphi)$, and $\mathcal{A} = \sigma(\mathcal{A}_1)$, then $\mathcal{A} + \mathcal{S} \in [\![I]\!]$. Therefore there exists $\mathcal{B} \in [\![I]\!]$ s.t. $\mathcal{B} = \mathcal{A} + \mathcal{S}$, i.e., there exists $\mathcal{B}_1 : \psi$, variant of an element in I, and an evaluation $\sigma' \in Sol(\psi)$ s.t. $\sigma'(\mathcal{B}_1) \preccurlyeq \mathcal{B}$.

  Let $\mathcal{B}' = \sigma'(\mathcal{B}_1) \backslash \mathcal{S}$, and $\mathcal{B}'_1 \preccurlyeq \mathcal{B}_1$ s.t. $\sigma'(\mathcal{B}'_1) = \mathcal{B}'$ (it is easy to show that such $\mathcal{B}'_1$ exists). By definitions and the relations $\sigma'(B_1) \preccurlyeq \mathcal{B}$ and $\mathcal{B} = \mathcal{A} + \mathcal{S}$, it also follows that $\mathcal{B}' \preccurlyeq \mathcal{A}$. Therefore, let $\mathcal{A}'_1 \preccurlyeq \mathcal{A}_1$ s.t. $\sigma(\mathcal{A}'_1) = \mathcal{B}'$.

  Now, let σ'' be the evaluation such that

  $$\sigma''_{|FV(\mathcal{B}_1,\psi)} = \sigma'_{|FV(\mathcal{B}_1,\psi)} \text{ and } \sigma'' = \sigma \text{ otherwise}$$

  (note that $\mathcal{B}_1 : \psi$ is a *variant*, therefore $\mathcal{B}_1 : \psi$ and $\mathcal{A}_1 : \varphi$ have no variables in common). Let $\varphi' \equiv \mathcal{B}'_1 = \mathcal{A}'_1 \wedge \varphi \wedge \psi$.

  We have that $\sigma'' \in Sol(\varphi')$, in fact $\sigma''(\mathcal{B}'_1) = \sigma'(\mathcal{B}'_1) = \mathcal{B}' = \sigma(\mathcal{A}'_1) = \sigma''(\mathcal{A}'_1)$, besides it can be easily shown that σ'' is a solution for both φ and ψ.

  It follows that $I \Vdash \mathcal{A}_1 : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$ and $\mathcal{C} = \mathcal{B}_1 \backslash \mathcal{B}'_1$. Therefore $\sigma''(\mathcal{C}) = \sigma''(\mathcal{B}_1 \backslash \mathcal{B}'_1) =$ (remember that $\mathcal{B}'_1 \preccurlyeq \mathcal{B}_1$) $\sigma''(\mathcal{B}_1) \backslash \sigma''(\mathcal{B}'_1) \preccurlyeq \mathcal{B} \backslash \mathcal{B}' = (\mathcal{A} + \mathcal{S}) \backslash \mathcal{B}' \preccurlyeq \mathcal{S}$ (because $\mathcal{B}' \preccurlyeq \mathcal{A}$). Finally, $\sigma''_{|FV(\mathcal{A}_1)} = \sigma_{|FV(\mathcal{A}_1)}$ by construction;
- if $[\![I]\!] \models F \,\&\, G, \Delta \blacktriangleright \mathcal{S}$, $\sigma \in Sol(\varphi)$, and $F \,\&\, G, \Delta = \sigma(F_1 \,\&\, G_1, \Delta_1)$, then $[\![I]\!] \models F, \Delta \blacktriangleright \mathcal{S}$, $[\![I]\!] \models G, \Delta \blacktriangleright \mathcal{S}$. By inductive hypothesis, there exist multisets $\mathcal{A}_1$ and $\mathcal{A}_2$, satisfiable constraints φ_1, φ_2, and evaluations $\sigma'_1 \in Sol(\varphi_1)$, $\sigma'_2 \in Sol(\varphi_2)$ s.t

  $$I \Vdash F_1, \Delta_1 : \varphi \blacktriangleright \mathcal{A}_1 \blacktriangleright \varphi_1 \text{ and } I \Vdash G_1, \Delta_1 : \varphi \blacktriangleright \mathcal{A}_2 \blacktriangleright \varphi_2,$$

  with $\sigma'_1(\mathcal{A}_1) \preccurlyeq \mathcal{S}$, $\sigma'_2(\mathcal{A}_2) \preccurlyeq \mathcal{S}$, $\sigma'_{1|FV(F_1,\Delta_1)} = \sigma_{|FV(F_1,\Delta_1)}$, and $\sigma'_{2|FV(G_1,\Delta_1)} = \sigma_{|FV(G_1,\Delta_1)}$. By Lemma 9.30, we also have that there exist ψ_1 and ψ_2 such that $\varphi_1 \equiv \varphi \wedge \psi_1$ and $\varphi_2 \equiv \varphi \wedge \psi_2$.

  Now, let σ'' be the evaluation s.t.

  $$\sigma''_{|FV(F_1,\Delta_1,\mathcal{A}_1,\varphi_1)} = \sigma'_{1|FV(F_1,\Delta_1,\mathcal{A}_1,\varphi_1)} \text{ and } \sigma'' = \sigma'_2 \text{ otherwise.}$$

  We have that σ'' is a solution for φ_1 (by construction) and σ'' is a solution for φ_2. In fact $\varphi_2 \equiv \varphi \wedge \psi_2$, σ'' is a solution for φ because $\varphi_1 \equiv \varphi \wedge \psi_1$, and σ'' is a solution for ψ_2 because $\sigma''_{|FV(\psi_2)} = \sigma'_{2|FV(\psi_2)}$ (note that $FV(\psi_2) \cap FV(\varphi) \subseteq FV(G_1, \Delta_1)$ by Lemma 9.30 and that σ'_1 and σ'_2 coincide on variables in $FV(G_1, \Delta_1) \cap FV(F_1, \Delta_1)$).

  Now, let $\mathcal{B} = \sigma''(\mathcal{A}_1) \bullet \sigma''(\mathcal{A}_2)$ (we recall that $\bullet$ denotes the *merge* of multisets,

see Section 2.2), and $\mathcal{A}'_1$, $\mathcal{A}'_2$ s.t. $\mathcal{A}'_1 \preccurlyeq \mathcal{A}_1$, $\mathcal{A}'_2 \preccurlyeq \mathcal{A}_2$, $\sigma''(\mathcal{A}'_1) = \sigma''(\mathcal{A}'_2)$, and $\sigma''(\mathcal{A}_1) + \sigma''(\mathcal{A}_2 \setminus \mathcal{A}'_2) = \mathcal{B}$ (it is easy to show that such $\mathcal{A}'_1$ and $\mathcal{A}'_2$ exist). We have that

$$I \Vdash F_1 \,\&\, G_1, \Delta : \varphi \blacktriangleright \mathcal{A} \blacktriangleright \varphi',$$

where $\varphi' \equiv \mathcal{A}'_1 = \mathcal{A}'_2 \wedge \varphi_1 \wedge \varphi_2$, $\sigma'' \in Sol(\varphi')$, and $\mathcal{A} = \mathcal{A}_1 + (\mathcal{A}_2 \setminus \mathcal{A}'_2)$. We also have that $\sigma''(\mathcal{A}) = \sigma''(\mathcal{A}_1 + (\mathcal{A}_2 \setminus \mathcal{A}'_2)) =$ (remember that $\mathcal{A}'_2 \preccurlyeq \mathcal{A}_2$) $\sigma''(\mathcal{A}_1) + \sigma''((\mathcal{A}_2 \setminus \mathcal{A}'_2)) = \mathcal{B} = \sigma''(\mathcal{A}_1) \bullet \sigma''(\mathcal{A}_2) \preccurlyeq \mathcal{S}$ because $\sigma''(\mathcal{A}_1) = \sigma'_1(\mathcal{A}_1) \preccurlyeq \mathcal{S}$ and $\sigma''(\mathcal{A}_2) = \sigma'_2(\mathcal{A}_2) \preccurlyeq \mathcal{S}$. Finally, ${\sigma''}_{|FV(F_1 \,\&\, G_1, \Delta)} = \sigma_{|FV(F_1 \,\&\, G_1, \Delta)}$ by construction;

- if $\Delta = G_1 \parr G_2, \Delta'$ of $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

The satisfiability judgment $\Vdash$ also satisfies the following properties.

Lemma 9.32 *For any interpretations I_1, I_2, ..., constrained multiset of goals $\Delta : \varphi$, fact $\mathcal{C}'$, constraint φ' and evaluation σ,*

i. if $I_1 \sqsubseteq I_2$, $I_1 \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$ and $\sigma \in Sol(\varphi')$, then there exist a fact $\mathcal{C}''$, a constraint φ'' and an evaluation $\sigma' \in Sol(\varphi'')$ s.t. $I_2 \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}'' \blacktriangleright \varphi''$, $\sigma'(\mathcal{C}'') \preccurlyeq \sigma(\mathcal{C}')$ and ${\sigma'}_{|FV(\Delta)} = \sigma_{|FV(\Delta)}$;

ii. if $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, $\bigsqcup_{i=1}^{\infty} I_i \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$ and $\sigma \in Sol(\varphi')$, then there exist $k \in \mathbb{N}$, a fact $\mathcal{C}''$, a constraint φ'' and an evaluation $\sigma' \in Sol(\varphi'')$ s.t. $I_k \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}'' \blacktriangleright \varphi''$, $\sigma'(\mathcal{C}'') \preccurlyeq \sigma(\mathcal{C}')$ and ${\sigma'}_{|FV(\Delta)} = \sigma_{|FV(\Delta)}$.

Proof

i. Suppose $I_1 \sqsubseteq I_2$ and $I_1 \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$. Then by Lemma 9.31 *i* we have that for every $\sigma \in Sol(\varphi')$, $[\![I_1]\!] \models \sigma(\Delta) \blacktriangleright \sigma(\mathcal{C}')$. By hypothesis, $[\![I_1]\!] \subseteq [\![I_2]\!]$, therefore by Lemma 7.6 *i* we have $[\![I_2]\!] \models \sigma(\Delta) \blacktriangleright \sigma(\mathcal{C}')$. The conclusion then follows from Lemma 9.31 *ii*;

ii. suppose $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ and $\bigsqcup_{i=1}^{\infty} I_i \Vdash \Delta : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$. Then by Lemma 9.31 *i* we have that for every $\sigma \in Sol(\varphi')$, $[\![\bigsqcup_{i=1}^{\infty} I_i]\!] \models \sigma(\Delta) \blacktriangleright \sigma(\mathcal{C}')$, i.e., as it can be readily verified from Definition 9.19 and Definition 9.20, $\bigcup_{i=1}^{\infty} [\![I_i]\!] \models \sigma(\Delta) \blacktriangleright \sigma(\mathcal{C}')$. By Lemma 7.6 *ii*, we have that there exists $k \in \mathbb{N}$ s.t. $[\![I_k]\!] \models \sigma(\Delta) \blacktriangleright \sigma(\mathcal{C}')$. The conclusion then follows from Lemma 9.31 *ii*.

□

We are now ready to define the abstract fixpoint operator $S_P : \mathcal{I} \to \mathcal{I}$. As in Chapter 7, we will proceed in two steps. We will first define an operator working over interpretations (i.e., elements of $\mathcal{P}(HB(P))$) and then we will lift it to our abstract domain $\mathcal{I}$ consisting of the equivalence classes of elements of $\mathcal{P}(HB(P))$ w.r.t. the relation $\simeq$ defined in Definition 9.19. Formally, we first introduce the following definitions.

Definition 9.33 (Clause Variants) *Let Π be a finite set of predicate symbols, and $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, Sol, \sqsubseteq^c \rangle$ a constraint system. Given an $LO(\mathcal{C})$ program P, the set of variants of clauses in P, denoted $Vrn(P)$, is defined as follows:*

$$Vrn(P) = \{(H \circ\!\!- G \;\square\; \varphi)\,\theta \mid \forall\, (H \circ\!\!- G \;\square\; \varphi) \in P \text{ and } \theta \text{ is a renaming of the variables in } FV(H \circ\!\!- G \;\square\; \varphi) \text{ with new variables}\}$$

Definition 9.34 (Symbolic Fixpoint Operator S_P) *Given an $LO(\mathcal{C})$ program P and an interpretation I, the symbolic fixpoint operator S_P is defined as follows:*

$$S_P(I) = \{\widehat{H} + \mathcal{C} : \exists \mathbf{x}.\varphi' \mid (H \circ\!\!- G \;\square\; \varphi) \in Vrn(P), \; I \Vdash G : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi' \text{ and } \mathbf{x} = FV(\varphi') \setminus FV(\widehat{H} + \mathcal{C})\}.$$

The following property shows that S_P is sound and complete w.r.t T_P.

Proposition 9.35 *Let P be an $LO(\mathcal{C})$ program and I and interpretation. Then $[\![S_P(I)]\!] = T_{Gnd(P)}([\![I]\!])$.*

Proof

- $[\![S_P(I)]\!] \subseteq T_{Gnd(P)}([\![I]\!])$.

Let $\mathcal{B} : \exists \mathbf{x}.\varphi' = \widehat{H} + \mathcal{C} : \exists \mathbf{x}.\varphi' \in S_P(I)$, where $H \circ\!\!- G \;\square\; \varphi$ is a variant of a clause in P, $I \Vdash G : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$, and $\mathbf{x} = FV(\varphi') \setminus FV(\mathcal{B})$.

Let $\sigma \in Sol(\exists \mathbf{x}.\varphi')$. Then, there exists $\sigma' \in Sol(\varphi')$ s.t. $\sigma'_{|FV(\mathcal{B})} = \sigma_{|FV(\mathcal{B})}$. By Lemma 9.30, there exists ψ s.t. $\varphi' \equiv \varphi \wedge \psi$, therefore $\sigma' \in Sol(\varphi)$.

Let $H_1 \circ\!\!- G_1 = \sigma'(H \circ\!\!- G) \in Gnd(P)$. By Lemma 9.31 i, we also have that $[\![I]\!] \models \sigma'(G) \blacktriangleright \mathcal{A}$ for all $\mathcal{A}$ s.t. $\sigma'(\mathcal{C}) \preccurlyeq \mathcal{A}$.

By T_P definition, $\widehat{H_1 + \mathcal{A}} \in T_{Gnd(P)}([\![I]\!])$, for all $\mathcal{A}$ s.t. $\sigma'(\mathcal{C}) \preccurlyeq \mathcal{A}$. That is, $\mathcal{D} \in T_{Gnd(P)}([\![I]\!])$, for all $\mathcal{D}$ s.t. $\sigma'(\mathcal{B}) \preccurlyeq \mathcal{D}$. Note that $\sigma'(\mathcal{B}) = \sigma(\mathcal{B})$, therefore it follows that $\mathcal{D} \in T_{Gnd(P)}([\![I]\!])$ for all $\mathcal{D} \in [\![\mathcal{B} : \exists \mathbf{x}\varphi']\!]$ and hence the conclusion.

- $T_{Gnd(P)}([\![I]\!]) \subseteq [\![S_P(I)]\!]$.

Let $\mathcal{B} = \widehat{H} + \mathcal{A} \in T_{Gnd(P)}([\![I]\!])$, where $H \circ\!\!- G \in T_{Gnd(P)}$ and $[\![I]\!] \models G \blacktriangleright \mathcal{A}$.

Then, there exist $H_1 \circ\!\!- G_1 \ \Box \ \varphi$ variant of a clause in P and $\sigma \in Sol(\varphi)$ s.t. $H \circ\!\!- G = \sigma(H_1 \circ\!\!- G_1)$. By Lemma 9.31 *ii* and Lemma 9.30, there exist a multiset $\mathcal{C}$, satisfiable constraints φ' and ψ, and evaluation σ' s.t. $\sigma' \in Sol(\varphi')$, ${\sigma'}_{|FV(G_1)} = \sigma_{|FV(G_1)}$, $\varphi' \equiv \varphi \wedge \psi$, $I \Vdash G_1 : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$, and $\sigma'(\mathcal{C}) \preccurlyeq \mathcal{A}$.

Let σ'' be the evaluation such that

$$ {\sigma''}_{|FV(H_1,\varphi)} = \sigma_{|FV(H_1,\varphi)} \ \text{ and } \ \sigma'' = \sigma' \ \text{ otherwise.} $$

We have that $\sigma'' \in Sol(\varphi')$. In fact, first of all $\varphi' \equiv \varphi \wedge \psi$; $\sigma'' \in Sol(\varphi)$ because $\sigma \in Sol(\varphi)$ and ${\sigma''}_{|FV(\varphi)} = \sigma_{|FV(\varphi)}$ by definition; finally, $\sigma'' \in Sol(\psi)$ because $\sigma' \in Sol(\psi)$ and ${\sigma''}_{|FV(\psi)} = {\sigma'}_{|FV(\psi)}$ (in fact, note that: $FV(\psi) \cap FV(H_1) \subseteq FV(G_1) \cup FV(\varphi)$ because H_1 does not appear in the judgment $I \Vdash G_1 : \varphi \blacktriangleright \mathcal{C} \blacktriangleright \varphi'$; $FV(\psi) \cap FV(\varphi) \subseteq FV(G_1)$ by Lemma 9.30; $\sigma_{|FV(G_1)} = {\sigma'}_{|FV(G_1)}$). We also have that $\sigma''(\mathcal{C}) = \sigma'(\mathcal{C})$ by Lemma 9.30 and σ'' definition.

Clearly, $\sigma'' \in Sol(\varphi')$ implies $\sigma'' \in Sol(\exists \mathbf{x}.\varphi')$. By S_P definition, $\widehat{H_1} + \mathcal{C} : \exists \mathbf{x}.\varphi' \in S_P(I)$. It follows that $\sigma''(\widehat{H_1} + \mathcal{C}) = \sigma''(\widehat{H_1}) + \sigma''(\mathcal{C}) = \sigma(\widehat{H_1}) + \sigma'(\mathcal{C}) = \widehat{H} + \sigma'(\mathcal{C}) \preccurlyeq \widehat{H} + \mathcal{A} = \mathcal{B}$, hence the conclusion. $\Box$

Furthermore, the following corollary holds.

Corollary 9.36 *For every LO($\mathcal{C}$) program P and interpretations I and J, if $I \simeq J$ then $S_P(I) \simeq S_P(J)$.*

Proof If $I \simeq J$, i.e., $[\![I]\!] = [\![J]\!]$, we have that $T_{Gnd(P)}([\![I]\!]) = T_{Gnd(P)}([\![J]\!])$, and, by Proposition 9.35, $[\![S_P(I)]\!] = [\![S_P(J)]\!]$, i.e., $S_P(I) \simeq S_P(J)$. $\Box$

The previous Corollary allows us to safely lift S_P definition to the lattice $\langle \mathcal{I}, \sqsubseteq \rangle$. Formally, we define the abstract fixpoint operator as follows.

Definition 9.37 (Abstract Fixpoint Operator S_P) *Given an LO($\mathcal{C}$) program P and an equivalence class $[I]_\simeq$ of $\mathcal{I}$, the abstract fixpoint operator S_P is defined as follows:*

$$ S_P([I]_\simeq) = [S_P(I)]_\simeq $$

where $S_P(I)$ is defined in Definition 9.34.

For the sake of simplicity, in the following we will often use I to denote its class $[I]_{\simeq}$, and we will simply use the term *(abstract) interpretation* to refer to an equivalence class, i.e., an element of $\mathcal{I}$. The abstract fixpoint operator S_P satisfies the following property.

Proposition 9.38 (Monotonicity and Continuity) *For every $LO(\mathcal{C})$ program P, the abstract fixpoint operator S_P is monotonic and continuous over the lattice $\langle \mathcal{I}, \sqsubseteq \rangle$.*

Proof

Monotonicity.

Assume that $I \sqsubseteq J$ and $\widehat{H} + \mathcal{C}' : \exists \mathbf{x}.\varphi' \in S_P(I)$, where $H \circ\!\!- G \ \square \ \varphi$ is a variant of a clause in P, $\mathbf{x} = FV(\varphi') \setminus FV(\widehat{H} + \mathcal{C}')$, and $I \Vdash G : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$.

Let $\sigma \in Sol(\exists \mathbf{x}.\varphi')$. Then, there exists $\tau \in Sol(\varphi')$ s.t. $\sigma_{|FV(H,C')} = \tau_{|FV(H,C')}$.

By Lemma 9.32 i, we have that there exist $\mathcal{C}''$, φ'' and $\tau' \in Sol(\varphi'')$ s.t. $J \Vdash G : \varphi \blacktriangleright \mathcal{C}'' \blacktriangleright \varphi''$, $\tau'(\mathcal{C}'') \preccurlyeq \tau(\mathcal{C}')$ and ${\tau'}_{|FV(G)} = \tau_{|FV(G)}$. By Lemma 9.30, we also have that $\tau \in Sol(\varphi)$ and there exists ψ s.t. $\varphi'' = \varphi \wedge \psi$.

By S_P definition, we have that $\widehat{H} + \mathcal{C}'' : \exists \mathbf{y}.\varphi'' \in S_P(J)$, with $\mathbf{y} = FV(\varphi'') \setminus FV(\widehat{H} + \mathcal{C}'')$.

Now, let σ' be the evaluation s.t.

$$ {\sigma'}_{|FV(H,\varphi)} = \tau_{|FV(H,\varphi)} \quad \text{and} \quad \sigma' = \tau' \quad \text{otherwise.} $$

We have that $\sigma' \in Sol(\varphi'')$. In fact, first of all $\varphi'' = \varphi \wedge \psi$; $\sigma' \in Sol(\varphi)$ because $\tau \in Sol(\varphi)$ and ${\sigma'}_{|FV(\varphi)} = \tau_{|FV(\varphi)}$ by definition; finally, $\sigma' \in Sol(\psi)$ because $\tau' \in Sol(\psi)$ and ${\sigma'}_{|FV(\psi)} = {\tau'}_{|FV(\psi)}$ (in fact, note that: $FV(\psi) \cap FV(H) \subseteq FV(G) \cup FV(\varphi)$ because H does not appear in the judgment $J \Vdash G : \varphi \blacktriangleright \mathcal{C}'' \blacktriangleright \varphi''$; $FV(\psi) \cap FV(\varphi) \subseteq FV(G)$ by Lemma 9.30; $\tau_{|FV(G)} = {\tau'}_{|FV(G)}$). We also have that $\sigma'(\mathcal{C}'') = \tau'(\mathcal{C}'')$ by Lemma 9.30 and σ' definition.

Clearly, $\sigma' \in Sol(\varphi'')$ implies $\sigma' \in Sol(\exists \mathbf{y}.\varphi'')$. Therefore we have proved that for every $\sigma \in Sol(\exists \mathbf{x}.\varphi')$ there exists $\sigma' \in Sol(\exists \mathbf{y}.\varphi'')$ s.t. $\sigma'(\widehat{H} + \mathcal{C}'') = \sigma'(\widehat{H}) + \sigma'(\mathcal{C}'') = \tau(\widehat{H}) + \tau'(\mathcal{C}'') \preccurlyeq \tau(\widehat{H}) + \tau(\mathcal{C}') = \tau(\widehat{H} + \mathcal{C}') = \sigma(\widehat{H} + \mathcal{C}')$.

Using the fact that $\widehat{H} + \mathcal{C}'' : \exists \mathbf{y}.\varphi'' \in S_P(J)$ and $\widehat{H} + \mathcal{C}' : \exists \mathbf{x}.\varphi' \in S_P(I)$, we can conclude that $[\![S_P(I)]\!] \subseteq [\![S_P(J)]\!]$, i.e., $S_P(I) \sqsubseteq S_P(J)$.

Continuity.

We show that S_P is finitary. Suppose $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, and $\widehat{H} + \mathcal{C}' : \exists \mathbf{x} \varphi' \in S_P(\bigsqcup_{i=1}^{\infty} I_i)$, with $H \circ\!\!- G \ \square \ \varphi$ variant of a clause in P, $\mathbf{x} = FV(\varphi') \setminus FV(\widehat{H} + \mathcal{C}')$, and $\bigsqcup_{i=1}^{\infty} I_i \Vdash G : \varphi \blacktriangleright \mathcal{C}' \blacktriangleright \varphi'$.

Let $\sigma \in Sol(\exists \mathbf{x}.\varphi')$. Then, there exists $\tau \in Sol(\varphi')$ s.t. $\sigma_{|FV(H,C')} = \tau_{|FV(H,C')}$.

By Lemma 9.32 *ii* we have that there exist $k \in \mathbb{N}$, $\mathcal{C}''$, φ'' and $\tau' \in Sol(\varphi'')$ s.t. $I_k \Vdash G : \varphi \blacktriangleright \mathcal{C}'' \blacktriangleright \varphi''$, $\tau'(\mathcal{C}'') \preccurlyeq \tau(\mathcal{C}')$, and $\tau'_{|FV(G)} = \tau_{|FV(G)}$.

By S_P definition, we have that $\widehat{H} + \mathcal{C}'' : \exists \mathbf{y}.\varphi'' \in S_P(I_k)$, with $\mathbf{y} = FV(\varphi'') \backslash FV(\widehat{H} + \mathcal{C}'')$.

At this point, we proceed exactly as above (proof of monotonicity) to build an evaluation $\sigma' \in Sol(\exists \mathbf{y}.\varphi'')$ s.t. $\sigma'(\widehat{H} + \mathcal{C}'') \preccurlyeq \sigma(\widehat{H} + \mathcal{C}')$.

It follows that $[\![S_P(\bigsqcup_{i=1}^{\infty} I_i)]\!] \subseteq [\![S_P(I_k)]\!]$. Using the fact that $[\![S_P(I_k)]\!] \subseteq [\![\bigsqcup_{i=1}^{\infty} S_P(I_i)]\!]$, we can conclude that $[\![S_P(\bigsqcup_{i=1}^{\infty} I_i)]\!] \subseteq [\![\bigsqcup_{i=1}^{\infty} S_P(I_i)]\!]$, i.e., $S_P(\bigsqcup_{i=1}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=1}^{\infty} S_P(I_i)$. □

Corollary 9.39 *For every LO($\mathcal{C}$) program P, $[\![lfp(S_P)]\!] = lfp(T_{Gnd(P)})$.*

Let $\mathcal{F}_{sym}(P) = lfp(S_P)$, then we have the following main theorem.

Theorem 9.40 (Soundness and Completeness) *For every LO($\mathcal{C}$) program P, $O(P) = [\![\mathcal{F}_{sym}(P)]\!]$.*

Proof From Definition 9.14, Theorem 7.10 and Corollary 9.39. □

The previous results give us an algorithm to compute the operational and fixpoint semantics of an LO($\mathcal{C}$) program P via the fixpoint operator S_P.

Example 9.41 Let $\Pi = \{p, q, r, s, t\}$, and consider unification constraints over the term language generated by a constant symbol a and a function symbol f. Let $\mathcal{V}$ be a denumerable set of variables and $x, y, z, \ldots \in \mathcal{V}$. Let $H \multimap G \;\square\; \varphi$ be the following clause

$$p(x) \parr p(z) \multimap (q(x') \parr r(y)) \,\&\, s(z') \;\square\; z = a \wedge x' = x \wedge z' = a$$

and I the interpretation of Example 9.29, i.e., consisting of the following constrained multisets:

$$\mathcal{M}_1 : \psi_1 \equiv t(u_1, v_1), q(w_1) : u_1 = f(q_1) \wedge w_1 = a$$
$$\mathcal{M}_2 : \psi_2 \equiv t(u_2, v_2), s(w_2) : v_2 = f(q_2) \wedge w_2 = a$$

From Example 9.29, we know that

$$I \Vdash G : \varphi \blacktriangleright t(u_1, v_1) \blacktriangleright \varphi' \quad \text{and} \quad I \Vdash G : \varphi \blacktriangleright t(u_1, v_1), t(u_2, v_2) \blacktriangleright \varphi'',$$

with $\varphi' \equiv u_1 = u_2 \wedge v_1 = v_2 \wedge \varphi_1 \wedge \varphi_2$, $\varphi'' \equiv \varphi_1 \wedge \varphi_2$,

$$\varphi_1 \equiv x' = w_1 \wedge w_1 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge u_1 = f(q_1),$$
$$\varphi_2 \equiv z' = w_2 \wedge w_2 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge v_2 = f(q_2).$$

Therefore, we get that $p(x), p(z), t(u_1, v_1) : \psi' \in S_P(I)$ and $p(x), p(z), t(u_1, v_1), t(u_2, v_2) : \psi'' \in S_P(I)$, where $\psi' \equiv \exists x', z', u_2, v_2, w_1, w_2, q_1, q_2.(\varphi') \equiv x = a \wedge z = a \wedge u_1 = f(q_1) \wedge v_1 = f(q_2)$ and $\psi'' \equiv \exists x', z', w_1, w_2, q_1, q_2.(\varphi'') \equiv x = a \wedge z = a \wedge u_1 = f(q_1) \wedge v_2 = f(q_2)$. Summarizing, $S_P(I)$ contains the two elements

$$p(x), p(z), t(u_1, v_1) : x = a \wedge z = a \wedge u_1 = f(q_1) \wedge v_1 = f(q_2)$$
$$p(x), p(z), t(u_1, v_1), t(u_2, v_2) : x = a \wedge z = a \wedge u_1 = f(q_1) \wedge v_2 = f(q_2)$$

By considering the ground instances of the above constrained multisets, we have that $[\![S_P(I)]\!]$ contains, e.g., $p(a), p(a), t(f(a), f(a))$ and $p(a), p(a), t(f(a), a), t(a, f(a))$. Note that different ways of choosing the sub-multisets $\mathcal{D}_1$ and $\mathcal{D}_2$ in Definition 9.28 yield different (and not redundant) elements. □

9.3 Ensuring Termination

In this section we present a fragment of LO, enriched with linear constraints over integer variables, for which termination of the bottom-up evaluation algorithm presented in Section 9.2 is guaranteed. We will exploit this result in Section 9.4. Specifically, we introduce the class of programs with *monadic* predicate symbols, over the constraint system NC of definition 9.5, and the corresponding class of constrained multisets. The results of this section appeared in [BD02]. We note, however, that some results presented in [BD02], specifically the ones concerning programs with predicate symbols with arity greater than one, were incorrect. Suitable further restrictions, which make the results of [BD02] sound, are currently under investigation.

Definition 9.42 (The $\mathcal{P}_1$(NC) class) *The class* $\mathcal{P}_1(\mathrm{NC})$ *consists of LO*(NC) *programs such that every predicate symbol has arity at most one.*

Definition 9.43 (Constrained Multisets $\mathcal{M}_1$(NC)) *The class* $\mathcal{M}_1(\mathrm{NC})$ *consists of constrained multisets with predicate symbols with arity at most one, annotated with* NC *constraints.*

Example 9.44 Let $\Pi = \{p, q, r, s\}$, with p, q having arity one and r, s arity zero. Let $\mathcal{V}$ be a denumerable set of variables, and $x, y, \ldots \in \mathcal{V}$, Then the clause

$$r \parr p(x) \parr q(y) \multimap s \parr p(x') \parr q(y') \sqsubset y = x \wedge x' = x \wedge y' > y$$

is in the class $\mathcal{P}_1(\mathrm{NC})$, and the constrained multiset $r, p(x), q(y), q(z) : x = y \wedge z > y$ is in the class $\mathcal{M}_1(\mathrm{NC})$. □

The following result holds.

Proposition 9.45 *The class $\mathcal{M}_1(\mathrm{NC})$ is closed under applications of S_P, i.e., if $I \subseteq \mathcal{M}_1(\mathrm{NC})$ then $S_P(I) \subseteq \mathcal{M}_1(\mathrm{NC})$.*

Proof Immediate by Definition 9.34 and Definition 9.28. Note that NC constraints are closed with respect to existential quantification. □

The other property we shall prove is that there exists an entailment relation for constrained multisets that ensures the termination of the fixpoint computation for the bottom-up semantics. To improve the presentation, we will prove this property in two steps.

Entailment for the class $\mathcal{M}_1(\mathrm{EC})$

We first consider the subclass $\mathcal{M}_1(\mathrm{EC})$ of constrained multisets annotated with equality constraints only (see Definition 9.6), i.e. without $>$-constraints. Without loss of generality, we assume hereafter to deal with a set of predicate symbols with arity *one*. If it is not the case, we can complete predicates with arity less than one with *dummy* variables.

Definition 9.46 *Let $\mathcal{M} : \varphi$ be a constrained multiset in $\mathcal{M}_1(\mathrm{EC})$. The symmetric and transitive closure of the relation $=$ induces an equivalence relation $\equiv$ on the variables in $\mathcal{M} : \varphi$. Let $C_1, \ldots, C_r$ be the $\equiv$-equivalence classes for these variables. We define M_i as the multiset of predicate symbols having a variable $x \in C_i$ as argument in $\mathcal{M}$. Furthermore, we define $S(\mathcal{M} : \varphi)$ as the multiset $\{M_1, \ldots, M_r\}$.*

Example 9.47 Let $\mathcal{M} : \varphi$ be the constrained multiset

$$p(x), q(y), p(z), q(t), r(u), q(v), r(w) : x = y \wedge x = z \wedge t = u \wedge v = w.$$

The variables x, y, z, t, u, v, w can be partitioned into three clusters $C_1 = \{x, y, z\}$, $C_2 = \{t, u\}$, and $C_3 = \{v, w\}$. $S(\mathcal{M} : \varphi)$ is the *multiset* consisting of the elements $M_1 = \{p, p, q\}$, $M_2 = \{q, r\}$, and $M_3 = \{q, r\}$, i.e. $S(\mathcal{M} : \varphi) = \{\{p, p, q\}, \{q, r\}, \{q, r\}\}$. □

We introduce now the following ordering $\sqsubseteq^S$ between multisets of *clusters* of predicate symbols.

Definition 9.48 *Let $\mathcal{N} : \psi$ and $\mathcal{M} : \varphi$ be two constrained multisets in $\mathcal{M}_1(\mathrm{EC})$, and let $S(\mathcal{N} : \psi) = \{N_1, N_2, \ldots, N_r\}$ and $S(\mathcal{M} : \varphi) = \{M_1, M_2, \ldots M_k\}$. We write $S(\mathcal{N} : \psi) \sqsubseteq^S S(\mathcal{M} : \varphi)$ iff there exists an injective mapping h from $\{1, \ldots, k\}$ to $\{1, \ldots, r\}$ such that $M_i \preccurlyeq N_{h(i)}$ for $i : 1, \ldots, k$.*

Example 9.49 As an example, $\{\{p,p,p\},\{t,t\},\{q,q\},\{r,r,r\}\} \sqsubseteq^S \{\{p,p\},\{q\},\{r,r\}\}$ by mapping $\{p,p\}$ into $\{p,p,p\}$ (in fact, $\{p,p\} \preccurlyeq \{p,p,p\}$), $\{q\}$ into $\{q,q\}$, and $\{r,r\}$ into $\{r,r,r\}$. □

The following property holds.

Proposition 9.50 *Let $\mathcal{N} : \psi$ and $\mathcal{M} : \varphi$ be two constrained multiset in $\mathcal{M}_1(\mathrm{EC})$. Then $S(\mathcal{N} : \psi) \sqsubseteq^S S(\mathcal{M} : \varphi)$ implies $[\![\mathcal{N} : \psi]\!] \subseteq [\![\mathcal{M} : \varphi]\!]$.*

Proof Let $S(\mathcal{N} : \psi) = \{N_1, N_2, \ldots, N_r\}$ and $S(\mathcal{M} : \varphi) = \{M_1, M_2, \ldots M_k\}$, and suppose $S(\mathcal{N} : \psi) \sqsubseteq^S S(\mathcal{M} : \varphi)$. Thus, for every M_i there exists a distinct $N_{h(i)}$ such that $M_i \preccurlyeq N_{h(i)}$. If $C \in [\![\mathcal{N} : \psi]\!]$, then there exist $\sigma \in Sol(\psi)$ and $\mathcal{D}$ s.t. then $C = \sigma(\mathcal{N}) + D$. Since $\sigma \in Sol(\psi)$, it follows that for $i : 1, \ldots, r$ there exists v_i s.t. $\sigma(x) = \sigma(y) = v_i$ for every pair of variables x and y in the i-th cluster of $\mathcal{N} : \psi$, i.e., $C = C_1 + \ldots + C_r + D$, where $C_i = \{p_{i1}(v_i), p_{i2}(v_i), \ldots, p_{im_i}(v_i)\}$ and $N_i = \{p_{i1}, p_{i2}, \ldots, p_{im_i}\}$ for $i : 1, \ldots, r$.

Let us now consider the multiset $E = E_1 + \ldots + E_k$ where $E_i = \{q_{i1}(v_{h(i)}), q_{i2}(v_{h(i)}), \ldots, q_{iz_i}(v_{h(i)})\}$, where $M_i = \{q_{i1}, q_{i2}, \ldots, q_{iz_i}\}$ for $i : 1, \ldots, k$. Then, by definition of $S(\mathcal{M} : \varphi)$, $E \in [\![\mathcal{M} : \varphi]\!]$. Furthermore, since, by hypothesis, there exist $N_{h(1)}, \ldots, N_{h(k)}$ s.t. $M_i \preccurlyeq N_{h(i)}$ for $i : 1, \ldots, k$, and from the injectivity of h, it follows that $E \preccurlyeq \sigma(\mathcal{N}) \preccurlyeq C$, hence $C \in [\![\mathcal{M} : \varphi]\!]$. □

Remark 9.51 Note that the condition stated in Proposition 9.50 is not *necessary*. For instance, consider the following counterexample: $\mathcal{M} : \varphi \equiv p(x), q(y) : true$ and $\mathcal{N} : \psi \equiv p(x'), q(y') : x' = y'$; clearly $[\![\mathcal{N} : \psi]\!] \subseteq [\![\mathcal{M} : \varphi]\!]$, but $\mathcal{N} : \psi \sqsubseteq^S \mathcal{M} : \varphi$ does not hold. In fact, we cannot find an *injective* mapping from $\{\{p\},\{q\}\}$, which contains two elements, into $\{\{pq\}\}$, which contains only one element.

Furthermore, we have the following property. In the following we use the notation $\sqsupseteq^S$ for the reverse of the relation $\sqsubseteq^S$, i.e. $S \sqsupseteq^S T$ stands for $T \sqsubseteq^S S$.

Lemma 9.52 *The entailment relation $\sqsupseteq^S$ between constrained multisets in $\mathcal{M}_1(\mathrm{EC})$ is a well-quasi-ordering.*

Proof The conclusion follows from Dickson's Lemma (see Proposition 6.25), stating that the submultiset relation for multisets over a finite set is a wqo, and from Proposition 6.24 iii). □

Entailment for the class $\mathcal{M}_1$(NC)

Let us now deal with the case of $>$-constraints. First of all, we note that, given a set S of constrained multisets in the class $\mathcal{M}_1$(NC), we can always construct a set S' of elements of the same class, such that:

- for every $\mathcal{M} : \varphi$ in S', every pair of variables x, y occurring in φ are related by an atomic constraint ($=$ or $>$) in φ;
- $[\![S]\!] = [\![S']\!]$.

The transformation of S into S' amounts to the *completion* of the constraints occurring in a constrained configuration with all the possible missing ones. Specifically, given a (satisfiable) constrained multiset $\mathcal{M} : \varphi$, we can transform it into the set of (satisfiable) constrained configurations obtained by completing the partial ordering induced by φ on the variables in $\mathcal{M}$ into a total ordering (if x is unrelated w.r.t. y in φ, we consider all the possible completions given by the alternatives $x = y$, $x > y$, or $y > x$).

Given a set S of constrained multisets in $\mathcal{M}_1$(NC), let us call $Comp(S)$ the operator that returns the (union of the sets of) completions of the elements in S. From now on, we will work using *completed* elements in the class $\mathcal{M}_1$(NC).

Let $\mathcal{M} : \varphi$ be a completed constrained multiset in $\mathcal{M}_1$(NC). As usual, the symmetric and transitive closure of the relation $=$ induces an equivalence relation $\equiv$ on the corresponding variables. Let $C_1, \ldots, C_r$ be the $\equiv$-equivalence classes for the variables of $\mathcal{M} : \varphi$.

We define M_i as the *multiset* of predicate symbols having a variable $x \in C_i$ as argument in $\mathcal{M}$. Since variables are totally ordered w.r.t. $>$ in φ (as we work with completed constrained multisets) and the constraint φ is *satisfiable*, we can order the resulting clusters $M_1, \ldots, M_r$ into a *string* $M_{i_1} \cdot \ldots \cdot M_{i_r}$ such that

- if p and q occur in M_{i_j}, then for some variables x and y, $p(x)$ and $q(y)$ occur in $\mathcal{M}$ and $x = y$ follows from φ;
- if p occurs in M_{i_j} and q occurs in M_{i_k}, with $i_j > i_k$, then for some variables x and y, $p(x)$ and $q(y)$ occur in $\mathcal{M}$ and $x > y$ follows from φ.

Based on this idea, we define $Str(\mathcal{M})$ as the string $M_{i_1} \cdot \ldots \cdot M_{i_r}$.

Example 9.53 Let $\mathcal{M} : \varphi$ be the constrained multiset

$$p(x), q(y), p(z), q(t), r(u) : x > y \wedge y = z \wedge z > t \wedge t = u.$$

The variables x, y, z, t, u can be partitioned into three clusters $C_1 = \{t, u\}$, $C_2 = \{y, z\}$, and $C_3 = \{x\}$. $Str(\mathcal{M} : \varphi)$ is the string $\{q, r\} \cdot \{p, q\} \cdot \{p\}$. □

We introduce now the following ordering between multisets of *clusters* of predicate symbols.

Definition 9.54 *Let $\mathcal{N} : \psi$ and $\mathcal{M} : \varphi$ be two constrained multisets in $\mathcal{M}_1(\mathrm{NC})$, and let $Str(\mathcal{N} : \psi) = N_1 \cdot N_2 \cdot \ldots \cdot N_r$ and $Str(\mathcal{M} : \varphi) = M_1 \cdot M_2 \cdot \ldots \cdot M_k$. We write $Str(\mathcal{N} : \psi) \sqsubseteq^* Str(\mathcal{M} : \varphi)$ iff there exists an injective mapping h from $1, \ldots, k$ to $1, \ldots, r$ such that if $i < j$ then $h(i) < h(j)$ (h is monotone), and $M_i \preccurlyeq N_{h(i)}$ for $i : 1, \ldots, k$.*

Example 9.55 As an example, $\{p, p, p\} \cdot \{t, t\} \cdot \{q, q\} \cdot \{r, r, r\} \sqsubseteq^S \{p, p\} \cdot \{q\} \cdot \{r, r\}$ by mapping $\{p, p\}$ into $\{p, p, p\}$ (in fact, $\{p, p\} \preccurlyeq \{p, p, p\}$), $\{q\}$ into $\{q, q\}$, and $\{r, r\}$ into $\{r, r, r\}$. On the contrary, $\{p, p, p\} \cdot \{t, t\} \cdot \{r, r, r\} \cdot \{q, q\} \sqsubseteq^S \{p, p\} \cdot \{q\} \cdot \{r, r\}$ does not hold (note that the previous mapping would not respect the *monotonicity* property). □

The following property holds.

Proposition 9.56 *Let $\mathcal{N} : \psi$ and $\mathcal{M} : \varphi$ be two constrained multisets in $\mathcal{M}_1(\mathrm{NC})$. Then $Str(\mathcal{N} : \psi) \sqsubseteq^* Str(\mathcal{M} : \varphi)$ implies $[\![\mathcal{N} : \psi]\!] \subseteq [\![\mathcal{M} : \varphi]\!]$.*

Proof Let $Str(\mathcal{N} : \psi) = N_1 \cdot N_2 \cdot \ldots \cdot N_r$ and $Str(\mathcal{M} : \varphi) = M_1 \cdot M_2 \cdot \ldots \cdot M_k$, and suppose $Str(\mathcal{N} : \psi) \sqsubseteq^* Str(\mathcal{M} : \varphi)$. Thus, for every M_i there exists a distinct $N_{h(i)}$ (with $h(i) < h(j)$ for $i < j$) such that $M_i \preccurlyeq N_{h(i)}$. If $C \in [\![\mathcal{N} : \psi]\!]$, then then there exist $\sigma \in Sol(\psi)$ and $\mathcal{D}$ s.t. $C = \sigma(\mathcal{N}) + D$. Since $\sigma \in Sol(\psi)$, it follows that for $i : 1, \ldots, r$ there exists v_i s.t. $\sigma(x) = \sigma(y) = v_i$ for every pair of variables x and y in the i-th cluster of $\mathcal{N} : \psi$, and $v_i > v_j$ for $i > j$. Thus, $C = C_1 + \ldots + C_r + D$, where $C_i = \{p_{i1}(v_i), p_{i2}(v_i), \ldots, p_{im_i}(v_i)\}$ and $N_i = \{p_{i1}, p_{i2}, \ldots, p_{im_i}\}$ for $i : 1, \ldots, r$, with $v_i > v_j$ if $i > j$.

Let us now consider the multiset $E = E_1 + \ldots + E_k$ where $E_i = \{q_{i1}(v_{h(i)}), q_{i2}(v_{h(i)}), \ldots, q_{iz_i}(v_{h(i)})\}$ and $M_i = \{q_{i1}, q_{i2}, \ldots, q_{iz_i}\}$ for $i : 1, \ldots, k$, with $v_i > v_j$ if $i > j$. Then, by definition of $Str(\mathcal{M} : \varphi)$ and from the monotonicity property of h, we have that $E \in [\![\mathcal{M} : \varphi]\!]$. Furthermore, since, by hypothesis, there exists $N_{h(1)}, \ldots, N_{h(k)}$ such that $M_i \preccurlyeq N_{h(i)}$ for $i : 1, \ldots, k$, and from the injectivity of h, it follows that $E \preccurlyeq \sigma(\mathcal{N}) \preccurlyeq C$, hence $C \in [\![\mathcal{M} : \varphi]\!]$. □

Furthermore, we have the following property. As usual, we denote by $\sqsupseteq^*$ the reverse of the relation $\sqsubseteq^*$.

Lemma 9.57 *The entailment relation $\sqsupseteq^*$ between completed constrained multisets in $\mathcal{M}_1(\mathrm{NC})$ is a well-quasi-ordering.*

Proof The conclusion follows from Dickson's Lemma (see Proposition 6.25) and Proposition 6.24 iv). □

We can therefore state the following result, which proves termination of the bottom-up evaluation for the class $\mathcal{P}_1(\mathrm{NC})$.

Corollary 9.58 *Let P be a $\mathcal{P}_1(\mathrm{NC})$ program. Then there exists $k \in \mathbb{N}$ such that $\mathcal{F}_{sym}(P) = \bigsqcup_{i=0}^{k} S_P{\uparrow_k}\,(\emptyset)$.*

Proof By Propositions 9.45, 9.57, and 6.23. □

Remark 9.59 Concerning Corollary 9.58, note that the termination of bottom-up evaluation has been proved as soon as the backward reachability algorithm discussed in Section 9.2 is extended so as to maintain intermediate constrained multisets in completed form. However, as completion of constrained multisets can be computationally expensive, efficiency reasons suggest to use entailment tests like, e.g., the $\sqsubseteq^m$ relation of Definition 9.22, which has been proved to be sound (though we have not directly proved termination with this ordering).

9.3.1 A Dynamic Abstraction from DC to NC

We conclude this section by discussing a *dynamic abstraction* (see Section 6.1.2.1) which can be used, in conjunction with the result given by Corollary 9.58, to automatically verify systems specified over the DC constraint system. An application of this abstraction will be presented in Section 9.4.2. The abstraction is as follows.

Definition 9.60 (Abstraction from DC to NC) *Let $\diamond \in \{=, >\}$. The abstraction α from* DC*-constraints to* NC*-constraints is defined as follows: $\alpha(true) = true$, $\alpha(false) = false$, $\alpha(\varphi_1 \wedge \varphi_2) = \alpha(\varphi_1) \wedge \alpha(\varphi_2)$, and*

$$\alpha(x \diamond y + c) = \begin{cases} x \diamond y \ \textit{if}\ c = 0 \\ x > y \ \textit{if}\ c > 0 \\ y > x \ \textit{if}\ c < 0 \ \textit{and}\ \diamond \ \textit{is}\ = \\ \textit{true otherwise} \end{cases}$$

The abstraction α can be lifted to constrained multisets and interpretations as follows: $\alpha(\mathcal{M} : \varphi) = \mathcal{M} : \alpha(\varphi)$, and $\alpha(I) = \{\alpha(\mathcal{M} : \varphi) \mid \mathcal{M} : \varphi \in I\}$.

Clearly, we have that $Sol(\varphi) \subseteq Sol(\alpha(\varphi))$ for every DC-constraint φ, and, consequently, $Sol(I) \subseteq Sol(\alpha(I))$ for every interpretation I.

We will see an example of application of this technique in Section 9.4.2.

9.4 An Example: The Ticket Protocol

We conclude this chapter with an example consisting in a *multi-client*, *multi-server parameterized* formulation of the classical *ticket* protocol. As usual, we have automatically validated the proposed examples using the verification tool of Appendix A.

The ticket protocol is a mutual exclusion protocol designed for multi-client systems operating on shared resources (e.g. memory). In order to access the critical section, every client executes the protocol, which is based on a first-in first-served access policy. A description of the protocol in an idealized high-level language is presented in Figure 9.2. We use $P \mid Q$ to denote the *interleaving parallel execution* of the sub-programs P and Q, and $\langle \cdot \rangle$ to denote atomic fragments of code. The initial number of clients is a parameter n of the protocol. The protocol works as follows. Initially, all clients are thinking, while t and s store the same initial value. When requesting the access to the critical section, a client stores the value of the current ticket t in its local variable a. A new ticket is then emitted by incrementing t. Clients wait for their turn until the value of their local variable a equals the value of s. After the elaboration inside the critical section, a process releases it and the current turn is updated by incrementing s. During the execution, the *global* state of the protocol consists of the internal state (current value of the local variable) of n processes together with the current value of s and t. As remarked in [BGP97], even with a *finite* number of clients (e.g. for $n = 2$) the values of the local variables of individual processes as well as s and t may get *unbounded*. This implies that any instance (for fixed values of n) of the scheme of Figure 9.2 gives rise to an infinite-state system. Obviously, the algorithm is supposed to work for any value of n.

In Section 9.4.1 we will discuss different formulations of the ticket protocol, in particular a *multi-client single-server* version and a *multi-client multi-server* version. We model it using LO clauses enriched with *difference constraints* (see 9.4) allowing arithmetic operations like increment and decrement of data variables. This way, we can give a model which is faithful to the original formulation of Figure 9.2: we do not abstract away global and local integer variables attached to individual clients, that in fact can still grow unboundedly in our specification.

Validation of the ticket protocol will be discussed in Section 9.4.2. Specifically, we have defined an *automated* and *conservative* abstraction that can be used to attack verification problems for systems defined over linear integer constraints (like the DC-constraints above) that lay outside the NC class for which we have proved that termination of backward reach-

THE SYSTEM WITH n PROCESSES

Program
global var $s, t : integer$
begin
$t := 0$
$s := 0$
$P_1 \mid \ldots \mid P_n$
end

THE i-th COMPONENT

Process P_i ::=
local var $a : integer$
repeat
$think:$ $\langle\ a := t;\ t := t + 1\ \rangle$
$wait:$ **when** $\langle\ a = s\ \rangle$ **do**
$use:$ **begin**
CRITICAL SECTION
$\langle\ s := s + 1\ \rangle$
end
forever

Figure 9.2: High-level description of the ticket protocol

ability is guaranteed. The abstraction maps linear integer constraints into NC-constraints, and it can be computed automatically on sets of constrained multisets. We will also discuss optimization of the fixpoint computation using *pruning* invariants.

9.4.1 Specifying the Ticket Protocol

In this sections we present different formulations of the ticket protocol. First of all, we have encoded a *multi-client*, *single-server* version of the protocol, i.e., with an arbitrary but finite number of clients, fixed after an initialization phase. Translation into LO(DC) is shown in Figure 9.3. The (infinite) collection of admissible initial states consists of all configurations with an *arbitrary* but finite number of thinking processes and two counters having the same initial value ($t = s$). This set can be specified via rules 1 and 2 in Figure 9.3: as usual we define a predicate called *init* acting as the seed of all possible runs of the protocol. The counters are represented via the atoms $count(t)$ and $turn(s)$. Thinking clients are represented via the propositional symbol *think* (the value of the local variable of a client does not matter at this point). The behaviour of an individual client can be described via rules 3 through 5, which faithfully correspond to the high-level description of Figure 9.2. The relations between the local variable and the global counters are represented via DC-constraints. Finally, we allow thinking processes to terminate their execution as specified by rule 6. Note that the specification is independent of the number of clients, and that we keep an explicit representation of the data variables without putting any restrictions on their values.

The second version of the ticket protocol is a slight variation of the one presented in Figure 9.3. Namely, we allow *dynamic* creation of clients. The extended model is obtained by introducing a *demon* process whose only goal is to non deterministically generate new

Initial States

1. $init \multimap think \parr init \quad \Box \quad true$
2. $init \multimap count(t) \parr turn(s) \quad \Box \quad t = s$

Individual Behaviour

3. $think \parr count(t) \multimap wait(a) \parr count(t') \quad \Box \quad a = t \wedge t' = t + 1$
4. $wait(a) \parr turn(s) \multimap use(a') \parr turn(s') \quad \Box \quad a = s \wedge a' = a \wedge s' = s$
5. $use(a) \parr turn(s) \multimap think \parr turn(s') \quad \Box \quad s' = s + 1$

Termination

6. $think \multimap \bot \quad \Box \quad true$

Figure 9.3: A multi-client, single-server version of the ticket protocol

clients. The encoding is obtained by changing the first two rules in Figure 9.3 with the rules below (whereas the rules for individual processes remain unchanged):

Initial States and Dynamic Process Generation

$1'.\quad init \multimap count(t) \parr turn(s) \parr demon \quad \Box \quad t = s$

$2'.\quad demon \multimap demon \parr think \quad \Box \quad true$

Finally, let us consider now a system with an *arbitrary* but finite number of *shared resources*, each one controlled by two local counters s and t. The idea is to associate a unique identifier to each resource and use it to stamp the corresponding pair of counters. The resulting specification is shown in Fig 9.4. We consider only the more interesting case in which clients and servers are generated dynamically. The process $demon(n)$ maintains a local counter n used to generate a new identifier, say id, to associate to newly created resources. Resources are in turn represented via pairs of the form $count(id, t)$ and $turn(id, s)$. A thinking process can *non-deterministically* chooses which resource to wait for by synchronizing with one of the counters in the system (rule 4 in Figure 9.4). After this choice, the algorithm behaves as usual w.r.t. to the chosen resource. The termination rules can be specified as natural extensions of the single-server case. We show an example trace in Figure 9.5, where P is the program in Figure 9.4. Note that in this specification the sources of infiniteness are: the number of clients, the number of shared resources, the values of resource identifiers and the values of tickets.

Initial States

1. $init \multimap demon(n) \;\Box\; true$

Dynamic Server and Process Generation

2. $demon(n) \multimap demon(n') \parr count(id,t) \parr turn(id',s) \;\Box\; n' = n+1 \wedge t = s \wedge id = n \wedge id' = id$
3. $demon(n) \multimap demon(n') \parr think \;\Box\; n' = n$

Individual Behaviour

4. $think \parr count(id,t) \multimap think(r) \parr count(id',t') \;\Box\; r = id \wedge id' = id \wedge t' = t$
5. $think(r) \parr count(id,t) \multimap wait(r',a) \parr count(id',t') \;\Box\; r = id \wedge a = t \wedge t' = t+1 \wedge r' = r \wedge id' = id$
6. $wait(r,a) \parr turn(id,s) \multimap use(r',a') \parr turn(id',s') \;\Box\; r = id \wedge a = s \wedge a' = a \wedge s' = s \wedge r' = r \wedge id' = id$
7. $use(r,a) \parr turn(id,s) \multimap think \parr turn(id',s') \;\Box\; r = id \wedge s' = s+1 \wedge id' = id$

Termination

8. $think(r) \multimap \bot \;\Box\; true$
9. $think \multimap \bot \;\Box\; true$

Figure 9.4: A multi-client, multi-server version of the ticket protocol

9.4.2 Verifying the Ticket Protocol

We have validated the different versions of the ticket protocol analyzed in Section 9.4.1, with respect to the mutual exclusion property, using the verification tool presented in Appendix A. Let us describe the verification process in detail.

Safety Properties. The set of violations to mutual exclusion can be represented through the following LO axioms:

$$use(x) \parr use(y) \multimap \top \;\Box\; true$$

for the single-server formulation, and

$$use(id,x) \parr use(id',y) \multimap \top \;\Box\; id = id'$$

for the multi-server formulation. In both cases, they denote all configurations with *at least* two clients in the critical section at the same time.

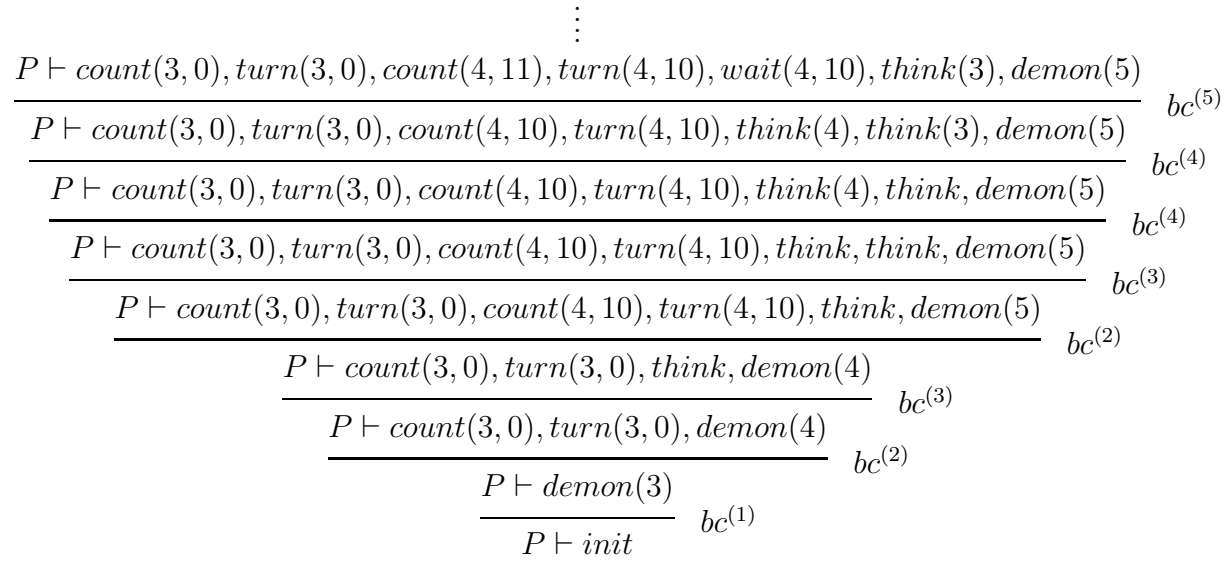

Figure 9.5: Multi-client ticket protocol: example trace

Relaxing Constraints. The bottom-up evaluation algorithm, when applied to the specifications in Figure 9.3 and Figure 9.4 and the above safety axioms, is not guaranteed to terminate. However, we can apply the abstraction technique described in Section 6.1.2.1 using the function α discussed in Section 9.3 (see Definition 9.60). In this way, termination is guaranteed for the specification given in Figure 9.3 and its variation based on dynamic process generation. In fact, the single-server specification of the ticket protocol of Figure 9.3 makes use of *monadic* predicate symbols, therefore using the abstraction α we have a verification problem covered by Corollary 9.58. Although the verification problem for the multi-server specification of Figure 9.4 is not covered by the results of Section 9.3, the symbolic backward reachability algorithm still terminates (see Table 9.1).

Pruning and Invariant Strengthening. Using the so-called *counting* abstraction, which simply forgets local data, while keeping track of the number of processes in a given state, we obtain models as expressive as Petri nets. Computing the structural invariants for the related Petri net model is not useful to prove properties that depend on the values attached to the tokens like mutual exclusion for the ticket protocol. However, these invariants can still be used in combination with the pruning technique of Section 6.1.2.2 in order to relieve the state explosion problem. Using a package for Petri net analysis, we can automatically obtain the invariants $x_{init} + x_{count} = 1$ and $x_{init} + x_{turn} = 1$, which imply that the number of tokens in places *turn* and *count* are always bounded by one. A similar reasoning can be applied to the multi-server protocol concerning the counters associated

Ticket Specification	Seed	α	Static Opt	Steps	Size	Time	Verified
Multi-client, Single-server	U_s			$\uparrow$	--	--	--
	U_s		*Pr/Invar*	$\uparrow$	--	--	--
	U_s	$\surd$		17	201	126	yes
	U_s	$\surd$	*Pruning*	9	23	< 1	yes
	U_s	$\surd$	*Invar*	9	26	< 1	yes
Multi-client, Single-server Dynamic Gen.	U_s			$\uparrow$	--	--	--
	U_s		*Pr/Invar*	$\uparrow$	--	--	--
	U_s	$\surd$		17	222	150	yes
	U_s	$\surd$	*Pruning*	10	32	< 1	yes
	U_s	$\surd$	*Invar*	10	34	< 1	yes
Multi-client, Multi-server Dynamic Gen.	U_m			$\uparrow$	--	--	--
	U_m		*Pr/Invar*	$\uparrow$	--	--	--
	U_m	$\surd$		> 18	> 3500	$> 4h$	--
	U_m	$\surd$	*Pruning*	19	141	15	yes
	U_m	$\surd$	*Invar*	19	147	19	yes

U_s is the singleton containing $use(x), use(y) : true$.
U_m is the singleton containing $use(id, x), use(id', y) : id = id'$.

Table 9.1: Validating the ticket protocol: experimental results

to a given identifier (at most one copy of each counter) and the demon process (at most one copy). We have used these information to prune the backward search space.

In an additional series of experiments, we have tested again the three models using the structural invariants to perform invariant strengthening (i.e., adding them to the set of unsafe states, see Section 6.1.2.1) instead of using them for pruning. This technique, so to say, has an effect which is similar to dynamic pruning (in fact, all constrained multisets that entail an invariant are discarded during the fixpoint computation, instead of being pruned).

Experimental Results. In Table 9.1, we show some experimental results. In column 'α', $\surd$ indicates that the abstraction α *has* been applied after each application of the symbolic predecessor operator; in column 'Static Opt' we have denoted by 'Pruning' and 'Invar' the use of structural invariants, for, respectively, pruning the search space or performing invariant strengthening ('Pr/Invar' denotes use of either technique); column 'Steps' contains the number of iterations needed to reach a fixpoint or before stopping the program ($\uparrow$ indicates that the procedure was still computing after several hours); column 'Size' denotes the number of constrained multisets contained in the fixpoint (or when the program was stopped); finally, 'Time' is the execution time (in seconds). We refer to Appendix A.4 for details on the experimental environment. As shown in Table 9.1, using the abstract (theoretically always terminating) backward reachability algorithm we managed to prove all

safety properties we were interested in. Furthermore, we managed to prove mutual exclusion without using structural invariants for the first two models, whereas it was necessary to use them in the third example, in order to avoid the state explosion problem.

9.5 Related Work

In this chapter we have defined a bottom-up semantics for the language LO enriched with constraints, from which we have derived a fully automated and sound method to attack verification of parameterized systems with unbounded local data. We have exemplified our approach by proving mutual exclusion for a parameterized, multi-client and multi-server formulation of the ticket protocol. The method is powered by using static analysis techniques coming from the structural theory of Petri Nets and by automatic abstractions working on constraints.

This work is inspired to the approach of [AJ98, AN00]. In [AJ98], Abdulla and Jonsson proposed an assertional language for Timed Petri Nets in which they use dedicated data structures to symbolically represent markings parametric in the number of tokens and in the *age* (represented as a real number) associated to tokens. In [AN00], Abdulla and Nylén formulate a symbolic algorithm using *existential regions* to represent the state-space of Timed Petri Nets. Our approach is an attempt to generalize the ideas of [AJ98, AN00] to problems and constraint systems that do not depend on the notion of *time*. In [AJ01a], Abdulla and Jonsson have used similar techniques to prove termination for backward reachability of *lossy/non lossy unordered channel systems* (i.e., finite-state control, unbounded channels) in which messages can vary over an infinite *name* domain.

For networks of *finite-state* processes, it is important to mention the automata theoretic approach to parameterized verification followed, e.g., in [KMM+97b, ABJN99, BBLS00, JN00, Nil00, PS00]. In this setting, the set of possible *local states* of individual processes are abstracted into a *finite alphabet*, whereas sets of global states are represented as networks parameterized by *regular languages*. Finite-state *transducers* are then applied to compute sets of predecessors. Manipulations on regular sets can be performed automatically by using tools like Mona [HJJ+95] or MoSel [KMM+97a]. In [KMM+97b], these ideas are applied to networks of processes arranged into an array topology, and specified in a second-order logic derived from WS1S (*weak second order logic of one successor* [Tho90]). An extension to tree topologies is considered. A similar approach is followed in [BBLS00], where symbolic exploration is performed by means of automated abstractions techniques. Specifically, networks of processes are specified in the logic WS1S and then abstracted into a *finite-state* system which can be model-checked. The approach can be used to verify a class of *liveness properties*. In [ABJN99], the focus is on model-checking of parameterized systems whose *actions* are specified by means of *local* transitions and *global* transitions.

Symbolic exploration is made effective by using operations over automata with ad hoc *accelerations*. Specifically, a method is devised to compute the result of applying a given action an arbitrary number of times. A general method for deriving the *transitive closure* for a large class of actions, improving the approach of [ABJN99], is presented in [JN00]. As a difference with [BBLS00], this method only works for *safety* properties. The construction of [ABJN99, JN00] is generalized in [PS00], where a model-checking procedure is presented, which can be used to verify *liveness* properties as well. Finally, a similar automata theoretic approach, called *regular model checking*, is presented in [Nil00] (see also [BJNT00]). It makes use of regular sets to represent both sets of states and transition relations, and it is usable for a class of liveness properties.

Though limited to studying safety properties, our approach has some advantages over the automata theoretic approaches described above. In particular, differently from the automata theoretic approach, in our setting we handle parameterized systems in which individual components have local variables that range over *unbounded* values. As an example, in our model of the ticket protocol local variables and tickets range over unbounded integers. Furthermore, note that the abstraction from DC to NC-constraints does not "finitize" the resulting symbolic representation, though termination can be guaranteed by applying the theory of well-quasi-orderings. This way, we do not have to apply *manual abstractions* to describe individual processes. This is an important aspect to take into account when comparing our practical results for the *single-server* system (the first experiment of Section 9.4.1) with those obtained in [JN00, Nil00], in which an *idealized* version of the *ticket algorithm* has been automatically verified using the regular model checking method (actually, a precise comparison is difficult here because the verified model is not described in [JN00, Nil00]).

The previous features also distinguish our approach from the *verification with invisible invariants* method of [PRZ01, APR$^+$01]. The method of [PRZ01, APR$^+$01] uses heuristics, based on model-checking *finite* instances of a parameterized system, to automatically *guess* induction invariants. Invariants computed in this way are then verified by checking they hold for a problem-dependent *finite* dimension of the parameterized system (a theorem is provided, working for some classes of invariants, which ensures that the invariants hold for every possible dimension). As an example, invisible invariants have been applied to automatically verify a *restricted* version of the parameterized *bakery* algorithm in which a special *reducing process* is needed to force the value of the tickets to stay within a given range (finite, though with parametric bound). A related approach is the one given in [FPP01], where the authors use constraint logic programming over the WSkS logic (*weak second order logic of k successors*) and *folding/unfolding* program transformations, to prove mutual exclusion for a parameterized version of the *bakery* algorithm.

As mentioned in the previous sections, a formulation of the *single-server* ticket protocol with two processes, but unbounded global and local variables, has been automatically

verified using constraint-based model checkers equipped with Presburger constraint solvers [BGP97], or *real arithmetic* constraint solvers [DP99]. We are not aware, however, of methods that can *automatically* handle the *parameterized*, *multi-server* and *multi-client* model of the ticket protocol presented in Section 9.4.1.

Our ideas are related to previous works connecting Constraint Logic Programming and verification, see e.g. [DP99, Fri99]. In this setting, transition systems are encoded via CLP programs used to encode the *global* state of a system and its updates. In our approach, we refine this idea by using multiset rewriting and constraints to *locally* specify updates to the *global* state. The notion of *constrained multiset* naturally extends the notion of *constrained atom* of [DP99]. The *locality* in the representation of rules allows us to consider *rich* denotations (upward-closures) instead of *flat* ones (instances) like, e.g., in [DP99]. This way, we can lift the approach to the parameterized case.

Finally, an alternative approach to verification is based on theorem proving, e.g. by using induction and invariant generation. We refer to Section 10.6 for a detailed discussion.

Summary of the Chapter. *In this chapter we have presented an extension of LO with constraints. This extension is inspired by traditional constraint programming languages, and is suitable to specify systems which use data structures with values ranging over heterogeneous domains. Validation can then be performed with the help of specialized constraint solvers. We have discussed the monadic fragment of LO, enriched with a subclass of linear integer constraints, for which we have proved termination of the bottom-up evaluation algorithm. We have also proved mutual exclusion for a parameterized, multi-client and multi-server formulation of the well-known ticket protocol.*

In the next chapter we will extend the language LO in another direction, namely we will consider a first-order formulation of the logic, with universal quantification in goals providing a way to generate new values.

Chapter 10

Bottom-Up Evaluation of First-Order LO Programs

In this chapter we will discuss a first-order formulation of LO which can be seen as a particular case of the construction shown in Chapter 9, where the constraint system at hand is the Herbrand constraint system HC of Definition 9.2. The first reason why we chose to deal with this fragment separately, is that the bottom-up evaluation construction of Chapter 9 can be specialized to the case of the Herbrand constraint system. Similarly to traditional logic programming, the specialized construction is based on notions like Herbrand interpretations, substitutions, and most general unifiers. The second reason for which we are discussing this fragment in a separate chapter, is that we want to allow an interesting extension, namely we want to extend LO programs with universal quantification in goals.

Universal quantification can be thought of as a primitive for dynamically introducing new *names* during the computation. To understand the point, consider the specification of the ticket protocol given in Figure 9.4, and in particular clause 2, given below.

$$2.\quad demon(n) \quad \circ\!\!- \quad demon(n') \parr count(id, t) \parr turn(id', s) \quad \Box \quad \begin{array}{l} n' = n + 1 \wedge t = s \wedge \\ id = n \wedge id' = id \end{array}$$

Here, we have used a $demon(n)$ process, where n works as a counter, to generate new *identifiers* for stamping pairs of *count*/*turn* processes. Given that we do not care about the actual value *count*/*turn* processes are stamped with (we only require that different pairs are associated with different values) we could achieve the same effect by exploiting the operational semantics of the universal quantifier (see Section 3.3.2). In the specific case, the only thing we have to do is to quantify over the variable id and use it to stamp both the *count* and the *turn* atoms (the counter n is not needed anymore). An example of specification showing the use of the universal quantifier will be presented in Section

10.5. Clearly, the connection between the bottom-up semantics for the fragment $\mathrm{LO}_\forall$ and verification of parameterized systems is the same as in Chapter 9. In particular, we will show that the fragment $\mathrm{LO}_\forall$ is suitable to reason about application domains like *security protocols* (this topic will be discussed in Chapter 11). In fact, the logical fragment considered in this chapter has been inspired by *multiset rewriting with universal quantification* (see Section 5.4) and [CDL+99], which discusses an equivalent fragment where *existential* quantification is used as a way to generate *nonces* in security protocols.

Technically, as in the previous chapters our aim is to define a bottom-up procedure to compute all goal formulas which are provable from a given program. Needless to say, the decidability property which we were able to prove for propositional LO is now lost. Nevertheless, we will still be able to define an *effective*, symbolic fixpoint operator for which every single step can be finitely computed, and we will able to prove soundness and completeness of this operator with respect to the operational semantics. Furthermore, re-using the results of Section 9.3, we will state termination of the bottom-up fixpoint computation algorithm for the class of *monadic* LO programs with universal quantification. In this chapter we will deal with the fragment $\mathrm{LO}_\forall$ (see Section 3.3.2), i.e., a first-order formulation of LO comprising the logical connectives $\parr$, $\multimap$, $\&$, $\top$, $\bot$, and universal quantification over goals.

As usual, in order to ease the proof of soundness and completeness, the definition of the bottom-up semantics will be presented in two steps. More precisely, we first present a simple, non-effective notion of (concrete) interpretation and the corresponding definition of fixpoint operator, which we call T_P. We then present an extended notion of (abstract) interpretation, and we define a symbolic and effective version of the fixpoint operator, called S_P. Owing to the presence of universal quantification, the semantic definition must carefully take into account the fact that program signatures can dynamically grow. As usual, for the sake of simplicity, we re-use some notations like, e.g., the ones for fixpoint operators and judgments.

10.1 A Proof-system for $\mathrm{LO}_\forall$

Some notations. Given an $\mathrm{LO}_\forall$ program P, we denote by Σ_P the signature comprising the set of constant, function, and predicate symbols in P. We assume to have an infinite set $\mathcal{V}$ of variable symbols (usually noted x, y, z, ...). In order to deal with signature augmentation (due to the presence of universal quantification over goals) we also need an infinite set E of new constants (called *eigenvariables*). We denote by Sig_P the set of signatures which comprise at least the symbols in Σ_P (and possibly some eigenvariables). $T_\Sigma^\mathcal{V}$ denotes the set of *non ground* terms over Σ and $A_\Sigma^\mathcal{V}$ the set of *non ground* atoms over Σ. Similarly to Chapter 7, multisets of atoms over $A_\Sigma^\mathcal{V}$ will be called *facts*, and usually noted

$\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... We will also overload the usual notation for sets to indicate multisets. The notation $\mathcal{B} \succcurlyeq \mathcal{A}$ will stand for $\mathcal{A} \preccurlyeq \mathcal{B}$. The notations for multisets of formulas (contexts) are the same as in Chapter 7. Some notions concerning substitutions and the definition of *most general unifiers* of multisets have been given in Section 2.4.

Before discussing the bottom-up semantics, we need to lift the definition of operational semantics to $\text{LO}_\forall$ programs. One possible solution is to resort to Definition 7.1 by considering the so-called *ground instances* of first-order programs. However, in presence of universal quantification in goals, this solution is not completely satisfactory. Consider, in fact, the following example. Take a signature with a predicate symbol p and two constants a and b, and consider the $\text{LO}_\forall$ program consisting of the axiom $\forall x.p(x) \circ\!\!- \top$ and the program consisting of the two axioms $p(a) \circ\!\!- \top$ and $p(b) \circ\!\!- \top$. The two programs have the same *ground* semantics according to Definition 7.1, i.e., $\{p(a), p(b)\}$. However, the $\text{LO}_\forall$ goal $\forall x.p(x)$ succeeds only in the first one, as the reader can verify. In order to distinguish the two programs, one possibility could be to consider the eigenvariables as part of the signature on which the ground semantics is computed. However, in this thesis we take an alternative approach, namely we consider the so-called *non ground* semantics. In particular, our aim in this chapter will be to extend the so-called *C-semantics* of [FLMP93, BGLM94] (see also Section 2.5) to first-order LO.

First of all, by analogy with Definition 9.33 we give the following definition.

Definition 10.1 (Clause Variants) *Given an $LO_\forall$ program P, the set of variants of clauses in P, denoted $Vrn(P)$, is defined as follows:*

$$Vrn(P) = \{(H \circ\!\!- G)\,\theta \mid \forall\,(H \circ\!\!- G) \in P \text{ and } \theta \text{ is a renaming of the variables in } FV(H \circ\!\!- G) \text{ with new variables}\}$$

Now, we need to reformulate the proof-theoretical semantics of Section 3.3.2 (see Figure 3.6). According to the C-semantics of [FLMP93, BGLM94], our goal is to define the set of *non ground* goals which are provable from a given program P with an *empty answer substitution*. Slightly departing from [FLMP93, BGLM94], we define the proof system presented in Figure 10.1. This proof system is based on the idea of considering a first order program as the (generally *infinite*) collection of (*non ground*) instances of its clauses. By *instance* of a clause $H \circ\!\!- G$, we mean a clause $H\theta \circ\!\!- G\theta$, where θ is *any* substitution. The reader can see that, with this intuition, the set of goals provable from the system presented in Figure 10.1 correspond to the set of non ground goals which are provable with an empty answer substitution according to [FLMP93, BGLM94]. We remark that in this proof system there is *no* notion of *unification*. This formulation of the proof system is the proof-theoretical counterpart of the bottom-up semantics we will define in Section 10.2.

All formulas (and also substitutions) on the right-hand side of sequents in Figure 10.1 are implicitly assumed to range over the set of *non ground* terms over Σ. Rule $\forall_r$ is responsible

$$\frac{}{P \vdash_\Sigma \top, \Delta}\ \top_r \qquad \frac{P \vdash_\Sigma G_1, G_2, \Delta}{P \vdash_\Sigma G_1 \parr G_2, \Delta}\ \parr_r \qquad \frac{P \vdash_\Sigma G_1, \Delta \quad P \vdash_\Sigma G_2, \Delta}{P \vdash_\Sigma G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash_\Sigma \Delta}{P \vdash_\Sigma \bot, \Delta}\ \bot_r \qquad \frac{P \vdash_{\Sigma,c} G[c/x], \Delta}{P \vdash_\Sigma \forall x.G, \Delta}\ \forall_r\ (c \notin \Sigma) \qquad \frac{P \vdash_\Sigma G\theta, \mathcal{A}}{P \vdash_\Sigma \widehat{H}\theta, \mathcal{A}}\ bc\ \ (H \circ\!\!- G\ \in Vrn(P))$$

Figure 10.1: A proof system for non ground semantics of $\mathrm{LO}_\forall$

for signature augmentation. Every time rule $\forall_r$ is fired, a new constant c is added to the current signature, and the resulting goal is proved in the new signature. This behaviour is standard in logic programming languages [MNPS91]. Rule *bc* denotes a backchaining (resolution) step, where θ indicates *any* substitution. For our purposes, we can assume $Dom(\theta) \subseteq FV(H) \cup FV(G)$ (we remind that $FV(F)$ denotes the *free* variables of F). Note that $H \circ\!\!- G$ is assumed to be a variant, therefore it has no variables in common with $\mathcal{A}$. According to the usual concept of *uniformity*, *bc* can be executed only if the right-hand side of the current sequent consists of atomic formulas. Rules $\top_r$, $\parr_r$, $\&_r$ and $\bot_r$ are the same as in propositional LO. A sequent is provable if all branches of its proof tree terminate with instances of the $\top_r$ axiom.

Clearly, the proof system of Figure 10.1 is not *effective*, however it will be sufficient for our purposes. An effective way to compute the set of goals which are provable from the above proof system will be discussed in Section 10.3.

We give the following definition, where $\vdash_\Sigma$ is the provability relation defined by the proof system in Figure 10.1.

Definition 10.2 (Operational Semantics) *Given an $LO_\forall$ program P, its operational semantics, denoted $O(P)$, is given by*

$$O(P) = \{\mathcal{A} \mid \mathcal{A} \textit{ is a multiset of (non ground) atoms in } A^{\mathcal{V}}_{\Sigma_P} \textit{ and } P \vdash_{\Sigma_P} \mathcal{A}\}.$$

Intuitively, the set $O(P)$ is closed by *instantiation*, i.e., $\mathcal{A}\theta \in O(P)$ for any substitution θ, provided $\mathcal{A} \in O(P)$. Note that the operational semantics only include multisets of (non ground) *atoms*, therefore no connective (including the *universal quantifier*) can appear in the set $O(P)$. However, the intuition will be that the variables appearing in a multiset in $O(P)$ must be implicitly considered *universally quantified* (e.g. $\{p(x), q(x)\} \in O(P)$ implies that the goal $\forall x.(p(x) \parr q(x))$ is provable from P). As usual, the idea is that provability of a compound goal can always be reduced to provability of a finite set of atomic multisets.

10.2 A Bottom-Up Semantics for $\mathbf{LO}_\forall$

We will now discuss the *bottom-up* semantics. In presence of universal quantification and therefore signature augmentation, we need to extend the definition of Herbrand base and (concrete) interpretations as follows. Namely, the definition of Herbrand base now depends explicitly on the signature, and interpretations can be thought of as infinite tuples, with one element for every signature $\Sigma \in Sig_P$. We give the following definitions.

Definition 10.3 (Herbrand Base) *Given an $LO_\forall$ program P and a signature $\Sigma \in Sig_P$, the Herbrand base of P over Σ, denoted $HB_\Sigma(P)$, is given by*

$$HB_\Sigma(P) = \mathcal{MS}(A^{\mathcal{V}}_\Sigma) = \{\mathcal{A} \mid \mathcal{A} \text{ is a multiset of (non ground) atoms in } A^{\mathcal{V}}_\Sigma\}.$$

Definition 10.4 (Interpretations) *Given an $LO_\forall$ program P, a (concrete) interpretation is a family of sets $\{I_\Sigma\}_{\Sigma \in Sig_P}$, where $I_\Sigma \in \mathcal{P}(HB_\Sigma(P))$ for every $\Sigma \in Sig_P$.*

In the following we often use the notation I for an interpretation to denote the family $\{I_\Sigma\}_{\Sigma \in Sig_P}$.

Interpretations form a complete lattice where inclusion and least upper bound are defined like (component-wise) set inclusion and union. In the following definition we therefore overload the symbols $\subseteq$ and $\cup$ for sets.

Definition 10.5 (Interpretation Domain) *Interpretations form a complete lattice $\langle \mathcal{D}, \subseteq \rangle$, where:*

- $\mathcal{D} = \{I \mid I \text{ is an interpretation}\}$;
- $I \subseteq J$ *iff* $I_\Sigma \subseteq J_\Sigma$ *for every* $\Sigma \in Sig_P$;
- *the least upper bound of* I *and* J *is* $\{I_\Sigma \cup J_\Sigma\}_{\Sigma \in Sig_P}$;
- *the bottom and top elements are* $\emptyset = \{\emptyset_\Sigma\}_{\Sigma \in Sig_P}$ *and* $\{HB_\Sigma(P)\}_{\Sigma \in Sig_P}$, *respectively.*

Before introducing the definition of fixpoint operator, we need to define the notion of satisfiability of a context Δ in a given interpretation I. For this purpose, we introduce the judgment $I \models_\Sigma \Delta \blacktriangleright \mathcal{C}$, where I is an interpretation, Δ is a context, and $\mathcal{C}$ is an output fact. The judgment is also parametric with respect to a given signature Σ. The parameter $\mathcal{C}$ must be thought of as an *output* fact such that $\mathcal{C} + \Delta$ is valid in I. The notion of output fact will simplify the presentation of the algorithmic version of the judgment which we will present in Section 10.3.

Remark 10.6 In what follows, using the notation $I \models_\Sigma \Delta \blacktriangleright \mathcal{C}$ we *always* make the *implicit* assumption that Δ is a context defined over Σ (i.e., terms constructors in Δ must belong to Σ). As a result, also the output fact $\mathcal{C}$ must be defined over Σ. This assumption, which is the counterpart of an analogous assumption for proof systems like the one in Figure 3.6, i.e., with explicit signature notation, will *always* and *tacitly* hold in the following. For example, note that in the $\forall$-case of the $\models_\Sigma$ definition below, the newly introduced constant c *cannot be exported* through the output fact $\mathcal{C}$. This is crucial to capture the operational semantics of the universal quantifier.

As usual, the notion of *satisfiability* is modeled according to the right-introduction (decomposition) rules of the proof system, as follows.

Definition 10.7 (Satisfiability Judgment) *Let P be an $LO_\forall$ program, $\Sigma \in Sig_P$, and $I = \{I_\Sigma\}_{\Sigma \in Sig_P}$ an interpretation. The satisfiability judgment $\models_\Sigma$ is defined as follows:*

$$
\begin{array}{l}
I \models_\Sigma \top, \Delta \blacktriangleright \mathcal{C} \text{ for any fact } \mathcal{C} \text{ in } A^{\mathcal{V}}_\Sigma; \\
I \models_\Sigma \mathcal{A} \blacktriangleright \mathcal{C} \text{ if } \mathcal{A} + \mathcal{C} \in I_\Sigma; \\
I \models_\Sigma \forall x.G, \Delta \blacktriangleright \mathcal{C} \text{ if } I \models_{\Sigma,c} G[c/x], \Delta \blacktriangleright \mathcal{C}, \text{ with } c \notin \Sigma \text{ (see remark 10.6)}; \\
I \models_\Sigma G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{C} \text{ if } I \models_\Sigma G_1, \Delta \blacktriangleright \mathcal{C}, \ I \models_\Sigma G_2, \Delta \blacktriangleright \mathcal{C}; \\
I \models_\Sigma G_1 \parr G_2, \Delta \blacktriangleright \mathcal{C} \text{ if } I \models_\Sigma G_1, G_2, \Delta \blacktriangleright \mathcal{C}; \\
I \models_\Sigma \bot, \Delta \blacktriangleright \mathcal{C} \text{ if } I \models_\Sigma \Delta \blacktriangleright \mathcal{C}.
\end{array}
$$

The satisfiability judgment $\models_\Sigma$ satisfies the following properties.

Lemma 10.8 *For every interpretation $I = \{I_\Sigma\}_{\Sigma \in Sig_P}$, context Δ, and fact $\mathcal{C}$,*

$$I \models_\Sigma \Delta \blacktriangleright \mathcal{C} \text{ iff } I \models_\Sigma \Delta, \mathcal{C} \blacktriangleright \epsilon.$$

Proof *If part.* By induction on the derivation of $I \models_\Sigma \Delta, \mathcal{C} \blacktriangleright \epsilon$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = \mathcal{A}$ and $\mathcal{A} + \mathcal{C} \in I_\Sigma$, then also $I \models_\Sigma \mathcal{A} \blacktriangleright \mathcal{C}$ holds;
- if $\Delta = \forall x.G, \Delta'$ and $I \models_{\Sigma,c} G[c/x], \Delta', \mathcal{C} \blacktriangleright \epsilon$, with $c \notin \Sigma$, then by inductive hypothesis $I \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, which implies $I \models_\Sigma \forall x.G, \Delta' \blacktriangleright \mathcal{C}$;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $I \models_\Sigma G_1, \Delta', \mathcal{C} \blacktriangleright \epsilon$ and $I \models_\Sigma G_2, \Delta', \mathcal{C} \blacktriangleright \epsilon$, by inductive hypothesis $I \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, which implies $I \models_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C}$;

- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

Only if part. By induction on the derivation of $I \models_\Sigma \Delta \blacktriangleright \mathcal{C}$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = \mathcal{A}$ and $\mathcal{A} + \mathcal{C} \in I_\Sigma$, then also $I \models_\Sigma \mathcal{A}, \mathcal{C} \blacktriangleright \epsilon$ holds;
- if $\Delta = \forall x.G, \Delta'$ and $I \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, with $c \notin \Sigma$, then by inductive hypothesis $I \models_{\Sigma,c} G[c/x], \Delta', \mathcal{C} \blacktriangleright \epsilon$, which implies $I \models_\Sigma \forall x.G, \Delta', \mathcal{C} \blacktriangleright \epsilon$;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $I \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, by inductive hypothesis $I \models_\Sigma G_1, \Delta', \mathcal{C} \blacktriangleright \epsilon$ and $I \models_\Sigma G_2, \Delta', \mathcal{C} \blacktriangleright \epsilon$, which implies $I \models_\Sigma G_1 \,\&\, G_2, \Delta', \mathcal{C} \blacktriangleright \epsilon$;
- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

Lemma 10.9 *For any interpretations* $I_1 = \{(I_1)_\Sigma\}_{\Sigma \in Sig_P}$, $I_2 = \{(I_2)_\Sigma\}_{\Sigma \in Sig_P}$, *. . . , context* Δ, *and fact* $\mathcal{C}$,

i. if $I_1 \subseteq I_2$ *and* $I_1 \models_\Sigma \Delta \blacktriangleright \mathcal{C}$ *then* $I_2 \models_\Sigma \Delta \blacktriangleright \mathcal{C}$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ *and* $\bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \blacktriangleright \mathcal{C}$ *then there exists* $k \in \mathbb{N}$ *s.t.* $I_k \models_\Sigma \Delta \blacktriangleright \mathcal{C}$.

Proof

i. By induction on the derivation of $I_1 \models_\Sigma \Delta \blacktriangleright \mathcal{C}$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = \mathcal{A}$ and $\mathcal{A} + \mathcal{C} \in (I_1)_\Sigma$, then $\mathcal{A} + \mathcal{C} \in (I_2)_\Sigma$, because $I_1 \subseteq I_2$, therefore $I_2 \models_\Sigma \mathcal{A} \blacktriangleright \mathcal{C}$;
- if $\Delta = \forall x.G, \Delta'$ and $I_1 \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, with $c \notin \Sigma$, then by inductive hypothesis $I_2 \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, which implies $I_2 \models_\Sigma \forall x.G, \Delta' \blacktriangleright \mathcal{C}$;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $I_1 \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I_1 \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, by inductive hypothesis $I_2 \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I_2 \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, which implies $I_2 \models_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C}$;
- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $\bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \blacktriangleright \mathcal{C}$.

- If $\Delta = \top, \Delta'$, then for every $k \in \mathbb{N}$, $I_k \models_\Sigma \Delta \blacktriangleright \mathcal{C}$;
- if $\Delta = \mathcal{A}$ and $\mathcal{A} + \mathcal{C} \in (\bigcup_{i=1}^{\infty} I_i)_\Sigma$, there exists $k \in \mathbb{N}$ s.t. $\mathcal{A} + \mathcal{C} \in (I_k)_\Sigma$, i.e., $I_k \models_\Sigma \mathcal{A} \blacktriangleright \mathcal{C}$;
- if $\Delta = \forall x.G, \Delta'$ and $\bigcup_{i=1}^{\infty} I_i \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, with $c \notin \Sigma$, then by inductive hypothesis there exists $k \in \mathbb{N}$ s.t. $I_k \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}$, therefore $I_k \models_\Sigma \forall x.G, \Delta' \blacktriangleright \mathcal{C}$;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, by inductive hypothesis there exist $k_1, k_2 \in \mathbb{N}$ s.t. $I_{k_1} \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I_{k_2} \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$. By taking $k = max\{k_1, k_2\}$, by i we get $I_k \models_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}$ and $I_k \models_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}$, which implies $I_k \models_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C}$;
- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

We are now ready to define the fixpoint operator T_P.

Definition 10.10 (Fixpoint Operator T_P) *Given an $LO_\forall$ program P and an interpretation $I = \{I_\Sigma\}_{\Sigma \in Sig_P}$, the fixpoint operator T_P is defined as follows:*

$$T_P(I) = \{(T_P(I))_\Sigma\}_{\Sigma \in Sig_P}$$

$$(T_P(I))_\Sigma = \{\widehat{H}\theta + \mathcal{C} \mid (H \multimap G) \in Vrn(P),\ \theta \text{ is any substitution, and } I \models_\Sigma G\theta \blacktriangleright \mathcal{C}\}.$$

Remark 10.11 In the previous definition, θ is implicitly assumed to be defined over Σ, i.e., θ can only map variables in $Dom(\theta)$ to terms in $T_\Sigma^{\mathcal{V}}$.

The following property holds.

Proposition 10.12 (Monotonicity and Continuity) *For every $LO_\forall$ program P, the fixpoint operator T_P is monotonic and continuous over the lattice $\langle \mathcal{D}, \subseteq \rangle$.*

Proof *Monotonicity.* Immediate from T_P definition and Lemma 10.9 *i*.
Continuity. We prove that T_P is finitary, i.e., for any increasing chain of interpretations $I_1 \subseteq I_2 \subseteq \ldots$ we have that $T_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$, i.e., for every $\Sigma \in \Sigma_P$, $(T_P(\bigcup_{i=1}^{\infty} I_i))_\Sigma \subseteq (\bigcup_{i=1}^{\infty} T_P(I_i))_\Sigma$. Let $\mathcal{A} \in (T_P(\bigcup_{i=1}^{\infty} I_i))_\Sigma$. By T_P definition, there exist $H \multimap G$ variant of a clause in P, a substitution θ and a fact $\mathcal{C}$ s.t. $\mathcal{A} = \widehat{H}\theta + \mathcal{C}$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G\theta \blacktriangleright \mathcal{C}$. By Lemma 10.9 *ii*, we have that there exists $k \in \mathbb{N}$ s.t. $I_k \models_\Sigma G\theta \blacktriangleright \mathcal{C}$. Again by T_P definition, we get $\mathcal{A} = \widehat{H}\theta + \mathcal{C} \in (T_P(I_k))_\Sigma \subseteq (\bigcup_{i=1}^{\infty} T_P(I_i))_\Sigma$. □

Monotonicity and continuity of the T_P operator imply, by Tarski's Theorem, that $lfp(T_P) = T_P{\uparrow}_\omega$. The fixpoint semantics of a program P is then defined as follows.

Definition 10.13 (Fixpoint Semantics) *Given an $LO_\forall$ program P, its fixpoint semantics, denoted $F(P)$, is defined as follows:*

$$F(P) = (lfp(T_P))_{\Sigma_P} = (T_P{\uparrow}_\omega(\{\emptyset_\Sigma\}_{\Sigma \in Sig_P}))_{\Sigma_P}.$$

We conclude this section by proving the following fundamental result, which states that the fixpoint semantics is sound and complete with respect to the operational semantics.

Theorem 10.14 (Soundness and Completeness) *For every $LO_\forall$ program P, $F(P) = O(P)$.*

Proof $F(P) \subseteq O(P)$. We prove that for every $k \in \mathbb{N}$, for every signature $\Sigma \in Sig_P$, and for every context Δ, $T_P{\uparrow}_k \models_\Sigma \Delta \blacktriangleright \epsilon$ implies $P \vdash_\Sigma \Delta$. The proof is by lexicographic induction on (k, h), where h is the length of the derivation of $T_P{\uparrow}_k \models_\Sigma \Delta \blacktriangleright \epsilon$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = \mathcal{A}$ and $\mathcal{A} \in (T_P{\uparrow}_k)_\Sigma$, then there exist a variant $H \circ\!\!- G$ of a clause in P, a fact $\mathcal{C}$ and a substitution θ s.t. $\mathcal{A} = \widehat{H}\theta + \mathcal{C}$ and $T_P{\uparrow}_{k-1} \models_\Sigma G\theta \blacktriangleright \mathcal{C}$. By Lemma 10.8, this implies $T_P{\uparrow}_{k-1} \models_\Sigma G\theta, \mathcal{C} \blacktriangleright \epsilon$. Then by inductive hypothesis we have $P \vdash_\Sigma G\theta, \mathcal{C}$, from which $P \vdash_\Sigma \widehat{H}\theta, \mathcal{C}$, i.e., $P \vdash_\Sigma \mathcal{A}$ follows by *bc* rule;
- if $\Delta = \forall x.G, \Delta'$ and $T_P{\uparrow}_k \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \epsilon$, with $c \notin \Sigma$, then by inductive hypothesis we have $P \vdash_{\Sigma,c} G[c/x], \Delta'$ from which $P \vdash_\Sigma \forall x.G, \Delta'$ follows by $\forall_r$ rule;
- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $T_P{\uparrow}_k \models_\Sigma G_1, \Delta' \blacktriangleright \epsilon$, and $T_P{\uparrow}_k \models_\Sigma G_2, \Delta' \blacktriangleright \epsilon$, then by inductive hypothesis we have $P \vdash_\Sigma G_1, \Delta'$ and $P \vdash_\Sigma G_2, \Delta'$, from which $P \vdash_\Sigma G_1 \,\&\, G_2, \Delta'$ follows by $\&_r$ rule;
- if $\Delta = G_1 \parr G_2, \Delta'$ and $T_P{\uparrow}_k \models_\Sigma G_1, G_2, \Delta' \blacktriangleright \epsilon$, then by inductive hypothesis we have $P \vdash_\Sigma G_1, G_2, \Delta'$, from which $P \vdash_\Sigma G_1 \parr G_2, \Delta'$ follows by $\parr_r$ rule;
- if $\Delta = \bot, \Delta'$ and $T_P{\uparrow}_k \models_\Sigma \Delta' \blacktriangleright \epsilon$, then by inductive hypothesis we have $P \vdash_\Sigma \Delta'$, from which $P \vdash_\Sigma \bot, \Delta'$ follows by $\bot_r$ rule.

$O(P) \subseteq F(P)$. We prove that for every signature $\Sigma \in Sig_P$ and for every context Δ, if $P \vdash_\Sigma \Delta$ then there exists $k \in \mathbb{N}$ s.t. $T_P{\uparrow}_k \models_\Sigma \Delta \blacktriangleright \epsilon$. The proof is by induction on the derivation of $P \vdash_\Sigma \Delta$.

- If $\Delta = \top, \Delta'$, then for every $k \in \mathbb{N}$, $T_P{\uparrow}_k \models_\Sigma \Delta \blacktriangleright \epsilon$;

- if $\Delta = \widehat{H}\theta, \mathcal{A}$, with $H \circ\!\!- G$ variant of a clause in P, θ substitution, and $P \vdash_\Sigma G\theta, \mathcal{A}$, then by inductive hypothesis we have that there exists $k \in \mathbb{N}$ s.t. $T_P{\uparrow}_k \models_\Sigma G\theta, \mathcal{A} \blacktriangleright \epsilon$. Then, by Lemma 10.8, $T_P{\uparrow}_k \models_\Sigma G\theta \blacktriangleright \mathcal{A}$. By T_P definition, $\widehat{H}\theta + \mathcal{A} \in (T_P{\uparrow}_{k+1})_\Sigma$, which implies $T_P{\uparrow}_{k+1} \models_\Sigma \widehat{H}\theta + \mathcal{A} \blacktriangleright \epsilon$;

- if $\Delta = \forall x.G, \Delta'$ and $P \vdash_{\Sigma,c} G[c/x], \Delta'$, with $c \notin \Sigma$, then by inductive hypothesis we have that there exist $k \in \mathbb{N}$ s.t. $T_P{\uparrow}_k \models_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \epsilon$, from which $T_P{\uparrow}_k \models_\Sigma \forall x.G, \Delta' \blacktriangleright \epsilon$ follows;

- if $\Delta = G_1 \,\&\, G_2, \Delta'$, $P \vdash_\Sigma G_1, \Delta'$ and $P \vdash_\Sigma G_2, \Delta'$, then by inductive hypothesis we have that there exist $k_1, k_2 \in \mathbb{N}$ s.t. $T_P{\uparrow}_{k_1} \models_\Sigma G_1, \Delta' \blacktriangleright \epsilon$ and $T_P{\uparrow}_{k_2} \models_\Sigma G_2, \Delta' \blacktriangleright \epsilon$. By taking $k = max\{k_1, k_2\}$, by Lemma 10.9 i and monotonicity of T_P we get $T_P{\uparrow}_k \models_\Sigma G_1, \Delta' \blacktriangleright \epsilon$ and $T_P{\uparrow}_k \models_\Sigma G_2, \Delta' \blacktriangleright \epsilon$, from which $T_P{\uparrow}_k \models_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \epsilon$ follows;

- if $\Delta = G_1 \parr G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

Example 10.15 Let Σ be a signature including the constant symbols a and b, a function symbol f, and the predicate symbols p, q, r, let $\mathcal{V}$ be a denumerable set of variables, and let P be the following following $\mathrm{LO}_\forall$ program:

1. $r(f(b)) \parr p(a) \circ\!\!- \top$
2. $p(x) \circ\!\!- \top$
3. $q(y) \circ\!\!- (\forall x.p(x)) \,\&\, r(y)$

Let $I_0 = \{\emptyset_\Sigma\}_{\Sigma \in Sig_P}$, and let us compute $I_1 = T_P(I_0)$. Using axioms 1 and 2, we get that $(I_1)_\Sigma$ contains the multisets of atoms of the form $\{r(f(b)), p(a)\} + \mathcal{A}$, and $\{p(t)\} + \mathcal{A}$, where $\mathcal{A}$ is any multiset of (possibly non-ground) atoms in $A^\mathcal{V}_\Sigma$, while t is any (possibly non ground) term in $T^\mathcal{V}_\Sigma$. Similarly $(I_1)_{\Sigma'}$, for a generic signature Σ' such that $\Sigma \subseteq \Sigma'$, contains all multisets of the above form where $\mathcal{A}$ and t are taken from, respectively, $A^\mathcal{V}_{\Sigma'}$ and $T^\mathcal{V}_{\Sigma'}$. For instance, let c be a new constant not appearing in Σ. The set $(I_1)_{\Sigma'}$ will contain, e.g., the multisets $\{p(c)\}$, $\{p(f(c)), q(b)\}$, and so on.

Now, consider the substitution $\theta = [y \mapsto f(b)]$ and the following corresponding instance of clause 3: $q(f(b)) \circ\!\!- (\forall x.p(x)) \,\&\, r(f(b))$. Suppose we want to compute an output fact $\mathcal{C}$ for the judgment

$$I_1 \models_\Sigma (\forall x.p(x)) \,\&\, r(f(b)) \blacktriangleright \mathcal{C}.$$

By $\models$ definition, we have to compute $I_1 \models_\Sigma (\forall x.p(x)) \blacktriangleright \mathcal{C}$ and $I_1 \models_\Sigma r(f(b)) \blacktriangleright \mathcal{C}$. For the latter judgment we have that, e.g., $I_1 \models_\Sigma r(f(b)) \blacktriangleright p(a)$. For the first judgment, by $\models$ definition, we must compute $I_1 \models_{\Sigma,c} \forall x.p(c) \blacktriangleright \mathcal{C}$, where c is a new constant not in Σ. As $\{p(c)\}$ is contained in $(I_1)_{\Sigma,c}$, we can get that $I_1 \models_{\Sigma,c} \forall x.p(c) \blacktriangleright \epsilon$. We can also get $I_1 \models_{\Sigma,c} \forall x.p(c) \blacktriangleright p(a)$ (in fact $\{p(c), p(a)\}$ is also contained in $(I_1)_{\Sigma,c}$. By applying the $\&$-rule for $\models$, we get that $I_1 \models_\Sigma (\forall x.p(x)) \,\&\, r(f(b)) \blacktriangleright p(a)$. Therefore, e.g., the multiset $\{q(b), p(a)\}$ is in $(I_2)_\Sigma = (T_P(I_1))_\Sigma$. □

10.3 An Effective Semantics for $\mathbf{LO}_\forall$

The fixpoint operator T_P defined in the previous section does not enjoy one of the crucial properties we required for our bottom-up semantics, namely its definition is *not* effective. This is a result of both the definition of the satisfiability judgment (whose clause for $\top$ is clearly not effective) and the definition of interpretations as infinite tuples. In order to solve these problems, we first define the (abstract) Herbrand base and (abstract) interpretations as follows.

Definition 10.16 (Abstract Herbrand Base) *Given an $LO_\forall$ program P, the Herband base of P, denoted $HB(P)$, is given by*

$$HB(P) = HB_{\Sigma_P}(P).$$

Definition 10.17 (Abstract Interpretations) *Given an $LO_\forall$ program P, an interpretation I is any subset of $HB(P)$, i.e., $I \in \mathcal{P}(HB(P))$.*

In order to define the abstract domain of interpretations, we need the following definitions.

Definition 10.18 (Instance Operator) *Given an interpretation I and a signature $\Sigma \in Sig_P$, we define the operator $Inst_\Sigma$ as follows:*

$$Inst_\Sigma(I) = \{\mathcal{A}\theta \mid \mathcal{A} \in I,\ \theta \text{ substitution over } \Sigma\}.$$

Definition 10.19 (Upward-closure Operator) *Given an interpretation I and a signature $\Sigma \in Sig_P$, we define the operator Up_Σ as follows:*

$$Up_\Sigma(I) = \{\mathcal{A} + \mathcal{C} \mid \mathcal{A} \in I,\ \mathcal{C} \text{ fact over } \Sigma\}.$$

Remark 10.20 Note that, as usual, in the previous definitions we assume the substitution θ and the fact $\mathcal{C}$ to be defined over Σ.

The following definition provides the connection between the (abstract) interpretations defined in Definition 10.17 and the (concrete) interpretations of Definition 10.4. The idea behind the definition is that an interpretation implicitly *denotes* the set of elements which can be obtained by either *instantiating* or *closing upwards* elements in the interpretation itself (where the concepts of instantiation and upward-closure are made precise by the above definitions). The operation of instantiation is related to the notion of C-semantics [FLMP93] (see Definition 10.2), while the operation of upward-closure is justified by Proposition 3.14. Note that the operations of instantiation and upward-closure are performed for every possible signature $\Sigma \in Sig_P$.

Definition 10.21 (Denotation of an Interpretation) *Given an (abstract) interpretation I, its denotation $[\![I]\!]$ is the (concrete) interpretation $\{[\![I]\!]_\Sigma\}_{\Sigma \in Sig_P}$ defined as follows:*

$$[\![I]\!]_\Sigma = Inst_\Sigma(Up_\Sigma(I)) \quad (\textit{or, equivalently, } [\![I]\!]_\Sigma = Up_\Sigma(Inst_\Sigma(I))).$$

Two interpretations I and J are said to be equivalent, written $I \simeq J$, if and only if $[\![I]\!] = [\![J]\!]$.

The equivalence of the two different equations in Definition 10.21 is stated in the following proposition.

Proposition 10.22 *For every interpretation I, and signature $\Sigma \in Sig_P$,*

$$Inst_\Sigma(Up_\Sigma(I)) = Up_\Sigma(Inst_\Sigma(I)).$$

Proof Let $(\mathcal{A} + \mathcal{C})\theta \in Inst_\Sigma(Up_\Sigma(I))$, with $\mathcal{A} \in I$. Then $(\mathcal{A} + \mathcal{C})\theta = (\mathcal{A}\theta) + \mathcal{C}\theta \in Up_\Sigma(Inst_\Sigma(I))$. Vice versa, let $\mathcal{A}\theta + \mathcal{C} \in Up_\Sigma(Inst_\Sigma(I))$, with $\mathcal{A} \in I$. Let $\mathcal{B}$ be a variant of $\mathcal{C}$ with new variables (not appearing in $\mathcal{A}$, θ, and $\mathcal{C}$) and θ' be the substitution with domain $Dom(\theta) \cup FV(\mathcal{B})$ and s.t. $\theta'_{|Dom(\theta)} = \theta$ and θ' maps $\mathcal{B}$ to $\mathcal{C}$. Then $\mathcal{A}\theta + \mathcal{C} = \mathcal{A}\theta' + \mathcal{B}\theta' = (\mathcal{A} + \mathcal{B})\theta' \in Inst_\Sigma(Up_\Sigma(I))$. □

We are now ready to define the abstract interpretation domain. As we do not need to distinguish between interpretations having the same denotation, we simply identify them using equivalence classes with respect to the corresponding equivalence relation $\simeq$.

Definition 10.23 (Abstract Interpretation Domain) *Abstract interpretations form a complete lattice $\langle \mathcal{I}, \sqsubseteq \rangle$, where*

- $\mathcal{I} = \{[I]_\simeq \mid I \textit{ is an interpretation}\}$;
- $[I]_\simeq \sqsubseteq [J]_\simeq$ *iff* $[\![I]\!] \subseteq [\![J]\!]$;

- *the least upper bound of* $[I]_\simeq$ *and* $[J]_\simeq$*, written* $[I]_\simeq \bigsqcup [J]_\simeq$*, is* $[I \cup J]_\simeq$*;*
- *the bottom and top elements are* $[\emptyset]_\simeq$ *and* $[\epsilon]_\simeq$*, respectively.*

The following proposition provides an *effective* and equivalent condition for testing the $\sqsubseteq$ relation over interpretations. We will need this result later on.

Proposition 10.24 *Given two interpretations* I *and* J*,* $[\![I]\!] \subseteq [\![J]\!]$ *iff for every* $\mathcal{A} \in I$*, there exist* $\mathcal{B} \in J$*, a substitution* θ *and a fact* $\mathcal{C}$ *(defined over* Σ_P*) s.t.* $\mathcal{A} = \mathcal{B}\theta + \mathcal{C}$*.*

Proof *If part.* We prove that for every $\Sigma \in Sig_P$, $[\![I]\!]_\Sigma \subseteq [\![J]\!]_\Sigma$. Let $\mathcal{A}' = \mathcal{A}\theta' + \mathcal{C}' \in Up_\Sigma(Inst_\Sigma(I)) = [\![I]\!]_\Sigma$, with $\mathcal{A} \in I$ and θ', $\mathcal{C}'$ defined over Σ. By hypothesis, there exist $\mathcal{B} \in J$, a substitution θ and a fact $\mathcal{C}$ (defined over Σ_P) s.t. $\mathcal{A} = \mathcal{B}\theta + \mathcal{C}$. Therefore, $\mathcal{A}' = \mathcal{A}\theta' + \mathcal{C}' = (\mathcal{B}\theta + \mathcal{C})\theta' + \mathcal{C}' = \mathcal{B}\theta\theta' + (\mathcal{C}\theta' + \mathcal{C}') \in Up_\Sigma(Inst_\Sigma(J)) = [\![J]\!]_\Sigma$ (note that $\theta\theta'$ and $\mathcal{C}\theta' + \mathcal{C}'$ are both defined over Σ because $\Sigma_P \subseteq \Sigma$).
Only if part. Let $\mathcal{A} \in I$, then $\mathcal{A} \in [\![I]\!]_{\Sigma_P}$ (note that $\mathcal{A}$ is defined over Σ_P by definition of interpretation). Then, by hypothesis we have that $\mathcal{A} \in [\![J]\!]_{\Sigma_P} = Up_{\Sigma_P}(Inst_{\Sigma_P}(J))$, i.e., there exist $\mathcal{B} \in J$, a substitution θ and a fact $\mathcal{C}$ (defined over Σ_P) s.t. $\mathcal{A} = \mathcal{B}\theta + \mathcal{C}$. □

We now define the abstract satisfiability judgment $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$, where I is an interpretation, Δ is a context, $\mathcal{C}$ is an output fact, and θ is an output substitution.

Remark 10.25 As usual, the notation $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ requires that Δ, $\mathcal{C}$, and θ are defined over Σ. As a consequence, the newly introduced constant c in the $\forall$-case of the $\Vdash_\Sigma$ definition below *cannot be exported* through the output parameters $\mathcal{C}$ or θ.

The judgment $\Vdash_\Sigma$ can be thought of as an abstract version of the judgment $\models_\Sigma$. We now need one more parameter, namely an output substitution. The idea behind the definition is that the output fact $\mathcal{C}$ and the output substitution θ are *minimal* (in a sense to be clarified) so that they can be computed effectively given a program P, an interpretation I, and a signature Σ. The output substitution θ is needed in order to deal with clause instantiation, and its minimality is ensured by using most general unifiers in the definition. We recall that the notation $\theta_1 \uparrow \theta_2$ denotes the least upper bound of substitutions (see Section 2.4).

Definition 10.26 (Abstract Satisfiability Judgment) *Let* P *be an* $LO_\forall$ *program,* I *an interpretation, and* $\Sigma \in Sig_P$*. The abstract satisfiability judgment* $\Vdash_\Sigma$ *is defined as*

follows:

$I \Vdash_\Sigma \top, \Delta \blacktriangleright \epsilon \blacktriangleright nil$;

$I \Vdash_\Sigma \mathcal{A} \blacktriangleright \mathcal{C} \blacktriangleright \theta$ *if there exist* $\mathcal{B} \in I$ *(variant),* $\mathcal{B}' \preccurlyeq \mathcal{B}$, $\mathcal{A}' \preccurlyeq \mathcal{A}$, $|\mathcal{B}'| = |\mathcal{A}'|$, $\mathcal{C} = \mathcal{B} \backslash \mathcal{B}'$, *and* $\theta = m.g.u.(\mathcal{B}', \mathcal{A}')_{|FV(\mathcal{A},\mathcal{C})}$;

$I \Vdash_\Sigma \forall x.G, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ *if* $I \Vdash_{\Sigma,c} G[c/x], \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$, *with* $c \notin \Sigma$ *(see remark 10.25)*;

$I \Vdash_\Sigma G_1 \,\&\, G_2, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ *if* $I \Vdash_\Sigma G_1, \Delta \blacktriangleright \mathcal{C}_1 \blacktriangleright \theta_1$, $I \Vdash_\Sigma G_2, \Delta \blacktriangleright \mathcal{C}_2 \blacktriangleright \theta_2$, $\mathcal{D}_1 \preccurlyeq \mathcal{C}_1$, $\mathcal{D}_2 \preccurlyeq \mathcal{C}_2$, $|\mathcal{D}_1| = |\mathcal{D}_2|$, $\theta_3 = m.g.u.(\mathcal{D}_1, \mathcal{D}_2)$, $\mathcal{C} = \mathcal{C}_1 + (\mathcal{C}_2 \backslash \mathcal{D}_2)$, *and* $\theta = (\theta_1 \uparrow \theta_2 \uparrow \theta_3)_{|FV(G_1,G_2,\Delta,\mathcal{C})}$;

$I \Vdash_\Sigma G_1 \parr G_2, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ *if* $I \Vdash_\Sigma G_1, G_2, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$;

$I \Vdash_\Sigma \bot, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ *if* $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$.

We recall that two multisets in general may have more than one (not necessarily equivalent) most general unifier and that using the notation $m.g.u.(\mathcal{B}', \mathcal{A}')$ we mean any unifier which is *non-deterministically* picked from the set of most general unifiers of $\mathcal{B}'$ and $\mathcal{A}'$ (see Section 2.4).

Example 10.27 Let us consider a signature with a function symbol f and predicate symbols p, q, r, s. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$. Let I be the interpretation consisting of the two multisets $\{p(x), q(x)\}$ and $\{r(y), p(f(y))\}$ (for simplicity, hereafter we omit brackets in multiset notation), and P the program

1. $r(w) \circ\!\!- q(f(w))$
2. $s(z) \circ\!\!- \forall x.p(f(x))$
3. $\bot \circ\!\!- q(u) \,\&\, r(v)$

Let's consider (a renaming of) the body of the first clause, $q(f(w'))$, and (a renaming of) the first element in I, $p(x'), q(x')$. Using the second clause for the $\Vdash_{\Sigma_P}$ judgment, with $\mathcal{A} = \mathcal{A}' = q(f(w'))$, $\mathcal{B} = p(x'), q(x')$, $\mathcal{B}' = q(x')$, we get

$$I \Vdash_{\Sigma_P} q(f(w')) \blacktriangleright p(x') \blacktriangleright [x' \mapsto f(w')].$$

Let's consider now (a renaming of) the body of the second clause, $\forall x.p(f(x))$, and another renaming of the first element, $p(x''), q(x'')$. From the $\forall$-case of $\Vdash_{\Sigma_P}$ definition, $I \Vdash_{\Sigma_P} \forall x.p(f(x)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$ if $I \Vdash_{\Sigma_P,c} p(f(c)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$, with $c \notin \Sigma_P$. Now, we can apply the second clause for $\Vdash_{\Sigma_P,c}$. Unfortunately, we can't choose $\mathcal{A}'$ to be $p(f(c))$ and $\mathcal{B}'$ to be $p(x'')$. In fact, by unifying $p(f(c))$ with $p(x'')$, we should get the substitution $\theta = [x'' \mapsto f(c)]$ and the output fact $q(x'')$ (note that x'' is a free variable in the output fact) and this is not allowed because the substitution θ must be defined on Σ_P, in order for $I \Vdash_{\Sigma_P} \forall x.p(f(x)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$

to be meaningful. It turns out that the only way to use the second clause for $\Vdash_{\Sigma_P,c}$ is to choose $\mathcal{A}' = \mathcal{B}' = \epsilon$, which is useless in the fixpoint computation (see Example 10.39).

Finally, let's consider (a renaming of) the body of the third clause, $\bot \circ\!\!- q(u') \,\&\, r(v')$. According to the $\&$-rule for the $\Vdash_{\Sigma_P}$ judgment, we must first compute $\mathcal{C}_1$, $\mathcal{C}_2$, θ_1 and θ_2 such that $I \Vdash_{\Sigma_P} q(u') \blacktriangleright \mathcal{C}_1 \blacktriangleright \theta_1$ and $I \Vdash_{\Sigma_P} r(v') \blacktriangleright \mathcal{C}_2 \blacktriangleright \theta_2$. To this aim, take two variants of the multisets in I, $p(x'''), q(x''')$ and $r(y'), p(f(y'))$. Proceeding as above, we get that

$$I \Vdash_{\Sigma_P} q(u') \blacktriangleright p(x''') \blacktriangleright [u' \mapsto x'''] \quad \text{and} \quad I \Vdash_{\Sigma_P} r(v') \blacktriangleright p(f(y')) \blacktriangleright [v' \mapsto y'].$$

Now, we can apply the $\&$-rule for the $\Vdash_{\Sigma_P}$ judgment, with $\mathcal{D}_1 = p(x''')$, $\mathcal{D}_2 = p(f(y'))$, and $\theta_3 = [x''' \mapsto f(y')]$. We have that $\theta_1 \uparrow \theta_2 \uparrow \theta_3 = [u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')]$. Therefore, we get that

$$I \Vdash_{\Sigma_P} q(u') \,\&\, r(v') \blacktriangleright p(x''') \blacktriangleright [u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')].$$

□

The following lemma states a simple property of the substitution domain, which we will need in the following.

Lemma 10.28 *For every interpretation I, context Δ, fact $\mathcal{C}$, and substitution θ, if $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ then $Dom(\theta) \subseteq FV(\Delta) \cup FV(\mathcal{C})$.*

Proof Immediate by induction on $\Vdash_\Sigma$ definition. □

The connection between the satisfiability judgments $\models_\Sigma$ and $\Vdash_\Sigma$ is clarified by the following lemma.

Lemma 10.29 *For every interpretation I, context Δ, fact $\mathcal{C}$, and substitution θ,*

i. if $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ then $[\![I]\!] \models_\Sigma \Delta\theta\theta' \blacktriangleright \mathcal{C}'\theta'$ for every substitution θ' and fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$;

ii. if $[\![I]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}$ then there exist a fact $\mathcal{C}'$, and substitutions θ' and σ s.t. $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(\Delta)} = (\theta' \circ \sigma)_{|FV(\Delta)}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$.

Proof

i. By induction on the derivation of $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$.

- If $\Delta = \top, \Delta'$, obvious;

- suppose $\Delta = \mathcal{A}$, with $\mathcal{B} \in I$ (variant), $\mathcal{B}' \preccurlyeq \mathcal{B}$, $\mathcal{A}' \preccurlyeq \mathcal{A}$, $\mathcal{C} = \mathcal{B} \setminus \mathcal{B}'$, and $\theta = m.g.u.(\mathcal{B}', \mathcal{A}')_{|FV(\mathcal{A},\mathcal{C})}$. We want to prove that $[\![I]\!] \models_\Sigma \mathcal{A}\theta\theta' \blacktriangleright \mathcal{C}'\theta'$ for every substitution θ' and fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$, i.e., $\mathcal{A}\theta\theta' + \mathcal{C}\theta\theta' + \mathcal{D}\theta' \in [\![I]\!]_\Sigma$ for every substitution θ' and fact $\mathcal{D}$.

  Now, $\mathcal{A}\theta\theta' + \mathcal{C}\theta\theta' + \mathcal{D}\theta' = (\mathcal{A}\theta + \mathcal{C}\theta + \mathcal{D})\theta' = (\mathcal{A}'\theta + (\mathcal{A}\setminus\mathcal{A}')\theta + (\mathcal{B}\setminus\mathcal{B}')\theta + \mathcal{D})\theta' =$ (remember that $\mathcal{B}' \preccurlyeq \mathcal{B}$) $(\mathcal{A}'\theta + (\mathcal{A} \setminus \mathcal{A}')\theta + (\mathcal{B}\theta \setminus \mathcal{B}'\theta) + \mathcal{D})\theta' = \mathcal{B}\theta\theta' + ((\mathcal{A} \setminus \mathcal{A}')\theta\theta' + \mathcal{D}\theta') \in [\![I]\!]_\Sigma$;
- if $\Delta = \forall x.G, \Delta'$ and $I \Vdash_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C} \blacktriangleright \theta$, with $c \notin \Sigma$, then by inductive hypothesis we have that

  $$[\![I]\!] \models_{\Sigma,c} G[c/x]\theta\theta', \Delta'\theta\theta' \blacktriangleright \mathcal{C}'\theta'$$

  for every substitution θ' and fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$ (where θ' and $\mathcal{C}'$ are defined over Σ, c).

  Assuming that the variable x is not in the domain of $\theta\theta'$ (it is always possible to rename the universally quantified variable x in $\forall x.G$), we have that $[\![I]\!] \models_{\Sigma,c} G\theta\theta'[c/x], \Delta'\theta\theta' \blacktriangleright \mathcal{C}'\theta'$, and, by definition of the judgment, we get $[\![I]\!] \models_\Sigma \forall x.(G\theta\theta'), \Delta'\theta\theta' \blacktriangleright \mathcal{C}'\theta'$, i.e., $[\![I]\!] \models_\Sigma (\forall x.G, \Delta')\theta\theta' \blacktriangleright \mathcal{C}'\theta'$, for every substitution θ' and fact $\mathcal{C}'$ defined over Σ, c (and therefore also for every substitution θ' and fact $\mathcal{C}'$ defined over Σ), with $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$;
- suppose $\Delta = G_1 \,\&\, G_2, \Delta'$ and $I \Vdash_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C} \blacktriangleright \theta$. We need to prove that $[\![I]\!] \models_\Sigma (G_1 \,\&\, G_2, \Delta')\theta\theta' \blacktriangleright \mathcal{C}'\theta'$ for every substitution θ' and fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$, i.e., that $[\![I]\!] \models_\Sigma (G_1 \,\&\, G_2, \Delta')\theta\theta' \blacktriangleright \mathcal{C}\theta\theta' + \mathcal{F}\theta'$ for every substitution θ' and fact $\mathcal{F}$.

  By $\Vdash_\Sigma$ definition, we have that there exist facts $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1, \mathcal{C}'_2 \preccurlyeq \mathcal{C}_2$ with $|\mathcal{C}'_1| = |\mathcal{C}'_2|$, and substitutions $\theta_1, \theta_2, \theta_3$ s.t.

  $$\theta_3 = m.g.u.(\mathcal{C}'_1, \mathcal{C}'_2), \quad \mathcal{C} = \mathcal{C}_1 + (\mathcal{C}_2 \setminus \mathcal{C}'_2), \quad \theta = (\theta_1 \uparrow \theta_2 \uparrow \theta_3)_{|FV(\Delta,\mathcal{C})},$$

  $$I \Vdash_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}_1 \blacktriangleright \theta_1 \;\text{ and }\; I \Vdash_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}_2 \blacktriangleright \theta_2.$$

  By inductive hypothesis, we have that

  $$[\![I]\!] \models_\Sigma (G_1, \Delta')\theta_1\theta'_1 \blacktriangleright \mathcal{C}_1\theta_1\theta'_1 + \mathcal{D}_1\theta'_1 \;\text{ and }\; [\![I]\!] \models_\Sigma (G_2, \Delta')\theta_2\theta'_2 \blacktriangleright \mathcal{C}_2\theta_2\theta'_2 + \mathcal{D}_2\theta'_2$$

  for every substitutions θ'_1, θ'_2 and facts $\mathcal{D}_1, \mathcal{D}_2$.

  By choosing $\mathcal{D}_1 = (\mathcal{C}_2 \setminus \mathcal{C}'_2)\theta_1 + \mathcal{F}_1$ and $\mathcal{D}_2 = (\mathcal{C}_1 \setminus \mathcal{C}'_1)\theta_2 + \mathcal{F}_2$, we have, for every substitutions θ'_1, θ'_2 and facts $\mathcal{F}_1, \mathcal{F}_2$,

  $$[\![I]\!] \models_\Sigma (G_1, \Delta')\theta_1\theta'_1 \blacktriangleright (\mathcal{C}_1 + (\mathcal{C}_2 \setminus \mathcal{C}'_2))\theta_1\theta'_1 + \mathcal{F}_1\theta'_1,$$
  $$[\![I]\!] \models_\Sigma (G_2, \Delta')\theta_2\theta'_2 \blacktriangleright (\mathcal{C}_2 + (\mathcal{C}_1 \setminus \mathcal{C}'_1))\theta_2\theta'_2 + \mathcal{F}_2\theta'_2.$$

By θ definition, we have that there exist substitutions $\gamma_1, \gamma_2, \gamma_3$ and τ s.t.

$$\tau = \theta_1 \circ \gamma_1, \quad \tau = \theta_2 \circ \gamma_2, \quad \tau = \theta_3 \circ \gamma_3, \quad \text{and} \quad \theta = \tau_{|FV(\Delta,\mathcal{C})}.$$

Now, let $\mathcal{F}_1$ be a variant of $\mathcal{F}\theta'$ with new variables, and define the substitution θ'_1 s.t. $Dom(\theta'_1) = Dom(\gamma_1 \circ \theta') \cup FV(\mathcal{F}_1)$ (clearly these two latter sets are disjoint), $\theta'_{1|Dom(\gamma_1 \circ \theta')} = \gamma_1 \circ \theta'$ and $\mathcal{F}_1\theta'_1 = \mathcal{F}\theta'$. Do the same for $\mathcal{F}_2$, i.e., let it be another variant of $\mathcal{F}\theta'$ with new variables, and define θ'_2 in the same way, so that $Dom(\theta'_2) = Dom(\gamma_2 \circ \theta') \cup FV(\mathcal{F}_2)$, $\theta'_{2|Dom(\gamma_2 \circ \theta')} = \gamma_2 \circ \theta'$, and $\mathcal{F}_2\theta'_2 = \mathcal{F}\theta'$.

Now, from τ definition it follows $(G_1, \Delta')\theta_1\theta'_1 = (G_1, \Delta')\theta_1\gamma_1\theta' = (G_1, \Delta')\theta\theta'$, and similarly $(G_2, \Delta')\theta_2\theta'_2 = (G_2, \Delta')\theta\theta'$. Besides, $(\mathcal{C}_1 + (\mathcal{C}_2 \backslash \mathcal{C}'_2))\theta_1\theta'_1 = \mathcal{C}\theta_1\theta'_1 = \mathcal{C}\theta\theta'$.

We also have that $(\mathcal{C}_2 + (\mathcal{C}_1 \backslash \mathcal{C}'_1))\theta_2\theta'_2 = (\mathcal{C}_2 + (\mathcal{C}_1 \backslash \mathcal{C}'_1))\theta_2\gamma_2\theta' = (\mathcal{C}_2 + (\mathcal{C}_1 \backslash \mathcal{C}'_1))\tau\theta' = (\mathcal{C}_2 + (\mathcal{C}_1 \setminus \mathcal{C}'_1))\theta_3\gamma_3\theta' =$ (remember that $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$) $(\mathcal{C}_2\theta_3 + (\mathcal{C}_1\theta_3 \setminus \mathcal{C}'_1\theta_3))\gamma_3\theta' =$ (remember that θ_3 is a unifier of $\mathcal{C}'_1$ and $\mathcal{C}'_2$) $(\mathcal{C}_2\theta_3 + (\mathcal{C}_1\theta_3 \setminus \mathcal{C}'_2\theta_3))\gamma_3\theta' =$ (note that $\mathcal{C}'_2\theta_3 = \mathcal{C}'_1\theta_3 \preccurlyeq \mathcal{C}_1\theta_3$) $((\mathcal{C}_2\theta_3 + \mathcal{C}_1\theta_3) \setminus \mathcal{C}'_2\theta_3)\gamma_3\theta' =$ (note that $\mathcal{C}'_2 \preccurlyeq \mathcal{C}_2$) $(\mathcal{C}_1\theta_3 + (\mathcal{C}_2\theta_3 \backslash \mathcal{C}'_2\theta_3))\gamma_3\theta' = (\mathcal{C}_1 + (\mathcal{C}_2 \backslash \mathcal{C}'_2))\theta_3\gamma_3\theta' = \mathcal{C}\theta_3\gamma_3\theta' = \mathcal{C}\theta\theta'$.

It follows that, by putting everything together, the inductive hypotheses become $[\![I]\!] \models_\Sigma (G_1, \Delta')\theta\theta' \blacktriangleright \mathcal{C}\theta\theta' + \mathcal{F}\theta'$ and $[\![I]\!] \models_\Sigma (G_2, \Delta')\theta\theta' \blacktriangleright \mathcal{C}\theta\theta' + \mathcal{F}\theta'$, from which the thesis follows by $\models_\Sigma$ definition;

- if $\Delta = G_1 \parr G_2, \Delta'$ and $I \Vdash_\Sigma G_1, G_2, \Delta' \blacktriangleright \mathcal{C} \blacktriangleright \theta$, then by inductive hypothesis we have that $[\![I]\!] \models_\Sigma (G_1, G_2, \Delta')\theta\theta' \blacktriangleright \mathcal{C}'\theta'$, for every substitution θ' and fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$. Therefore, $[\![I]\!] \models_\Sigma G_1\theta\theta', G_2\theta\theta', \Delta'\theta\theta' \blacktriangleright \mathcal{C}'\theta'$, and, by definition of the judgment, we get $[\![I]\!] \models_\Sigma (G_1 \parr G_2, \Delta)\theta\theta' \blacktriangleright \mathcal{C}'\theta'$;
- if $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[\![I]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}$.

- If $\Delta = \top, \Delta'$, take $\mathcal{C}' = \epsilon$, $\theta' = nil$, and $\sigma = \theta$;
- suppose $[\![I]\!] \models_\Sigma \mathcal{A}\theta \blacktriangleright \mathcal{C}$ and $\mathcal{A}\theta + \mathcal{C} \in [\![I]\!]_\Sigma = Up_\Sigma(Inst_\Sigma(I))$. Then there exist $\mathcal{B} \in I$, a fact $\mathcal{D}$ and a substitution τ (defined on Σ) s.t. $\mathcal{A}\theta + \mathcal{C} = \mathcal{B}\tau + \mathcal{D}$. We can safely assume, thanks to the substitution τ, that $\mathcal{B}$ is a *variant* of an element in I. Also, we can assume that $Dom(\tau) \subseteq FV(\mathcal{B})$ and $Dom(\theta) \cap Dom(\tau) = \emptyset$.

  Now, take the substitution γ s.t. $Dom(\gamma) = (Dom(\theta) \cap FV(\mathcal{A})) \cup Dom(\tau)$,

  $$\gamma_{|Dom(\theta) \cap FV(\mathcal{A})} = \theta_{|Dom(\theta) \cap FV(\mathcal{A})} \quad \text{and} \quad \gamma_{|Dom(\tau)} = \tau.$$

  We have that $\mathcal{A}\gamma + \mathcal{C} = \mathcal{B}\gamma + \mathcal{D}$. Let $\mathcal{A}' \preccurlyeq \mathcal{A}$ and $\mathcal{B}' \preccurlyeq \mathcal{B}$ be two *maximal* sub-multisets s.t. $\mathcal{A}'\gamma = \mathcal{B}'\gamma$, $\rho = m.g.u.(\mathcal{A}', \mathcal{B}')$, and $\theta' = \rho_{|FV(\mathcal{A}) \cup FV(\mathcal{B}\mathcal{B}')}$. By

definition of the $\Vdash_\Sigma$ judgment, we have that $I \Vdash_\Sigma \mathcal{A} \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, where $\mathcal{C}' = \mathcal{B} \backslash \mathcal{B}'$.

As γ is a unifier for $\mathcal{A}'$,$\mathcal{B}'$, while $\rho = m.g.u.(\mathcal{A}', \mathcal{B}')$, we have that there exists a substitution σ s.t. $\gamma = \rho \circ \sigma$. Therefore, $\theta_{|FV(\mathcal{A})} = \gamma_{|FV(\mathcal{A})} = (\rho \circ \sigma)_{|FV(\mathcal{A})} = (\rho_{|(FV(\mathcal{A}) \cup FV(\mathcal{B} \backslash \mathcal{B}'))} \circ \sigma)_{|FV(\mathcal{A})} = (\theta' \circ \sigma)_{|FV(\mathcal{A})}$, as required.

Furthermore, since $\mathcal{A}\gamma + \mathcal{C} = \mathcal{B}\gamma + \mathcal{D}$ and $\mathcal{A}' \preccurlyeq \mathcal{A}$, it follows that $\mathcal{A}'\gamma + (\mathcal{A} \setminus \mathcal{A}')\gamma + \mathcal{C} = \mathcal{B}'\gamma + (\mathcal{B} \setminus \mathcal{B}')\gamma + \mathcal{D}$, i.e., $(\mathcal{A} \setminus \mathcal{A}')\gamma + \mathcal{C} = (\mathcal{B} \setminus \mathcal{B}')\gamma + \mathcal{D}$. By this equality and maximality of $\mathcal{A}'$ and $\mathcal{B}'$, we get that necessarily $(\mathcal{B} \setminus \mathcal{B}')\gamma \preccurlyeq \mathcal{C}$ (otherwise, $(\mathcal{B} \backslash \mathcal{B}')\gamma$ and $(\mathcal{A} \backslash \mathcal{A}')\gamma$ would have elements in common). Therefore, $\mathcal{C}'\theta'\sigma = (\mathcal{B} \backslash \mathcal{B}')\theta'\sigma = (\mathcal{B} \backslash \mathcal{B}')\rho\sigma = (\mathcal{B} \backslash \mathcal{B}')\gamma \preccurlyeq \mathcal{C}$, as required;

- if $\Delta = \forall x.G, \Delta'$ and $[\![I]\!] \models_{\Sigma,c} (G[c/x], \Delta')\theta \blacktriangleright \mathcal{C}$, with $c \notin \Sigma$, then by inductive hypothesis there exist a fact $\mathcal{C}'$, and substitutions θ' and σ (defined over Σ, c) s.t.

$$I \Vdash_{\Sigma,c} G[c/x], \Delta' \blacktriangleright \mathcal{C}' \blacktriangleright \theta',$$

$\theta_{|FV(G[c/x],\Delta')} = (\theta' \circ \sigma)_{|FV(G[c/x],\Delta')}$, and $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$. By definition of the $\Vdash_\Sigma$ judgment, we get that

$$I \Vdash_\Sigma \forall x.G, \Delta' \blacktriangleright \mathcal{C}' \blacktriangleright \theta'.$$

The conclusion follows (remember that we must ensure that $\mathcal{C}'$, θ' and σ are defined over Σ) by the following crucial observations:

  - $Dom(\theta') \subseteq (FV(G[c/x], \Delta') \cup FV(\mathcal{C}'))$ by Lemma 10.28;
  - θ' does not map variables in $G[c/x], \Delta'$ to the eigenvariable c. In fact we know that θ does not map variables in $G[c/x], \Delta'$ to c (by hypothesis) and we know that $(\theta' \circ \sigma)_{|FV(G[c/x],\Delta')} = \theta_{|FV(G[c/x],\Delta')}$;
  - θ' does not map variables in $\mathcal{C}'$ to c and $\mathcal{C}'$ itself does not contain c. In fact we know that $\mathcal{C}$ does not contain c (by hypothesis) and also that $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$;
  - we can safely assume that $Dom(\sigma)$ does not contain variables mapped to c. Intuitively, these bindings are useless. Formally, we can restrict the domain of σ to variables that are not mapped to c: with this restriction, the equalities $\theta_{|FV(G[c/x],\Delta')} = (\theta' \circ \sigma)_{|FV(G[c/x],\Delta')}$ and $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$ still hold.

- suppose $\Delta = G_1 \,\&\, G_2, \Delta'$ and $[\![I]\!] \models_\Sigma (G_1 \,\&\, G_2\Delta')\theta \blacktriangleright \mathcal{C}$. We need to prove that there exist a fact $\mathcal{C}'$ and substitutions θ' and σ s.t. $I \Vdash_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(G_1,G_2,\Delta')} = (\theta' \circ \sigma)_{|FV(G_1,G_2,\Delta')}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$. By $\models_\Sigma$ definition, we have that

$$I \models_\Sigma (G_1, \Delta')\theta \blacktriangleright \mathcal{C} \;\text{ and }\; I \models_\Sigma (G_2, \Delta')\theta \blacktriangleright \mathcal{C}.$$

By inductive hypothesis, we have that there exist facts $\mathcal{C}_1, \mathcal{C}_2$ and substitutions θ_1, θ_2, σ_1, σ_2 s.t.

$$I \Vdash_\Sigma G_1, \Delta' \blacktriangleright \mathcal{C}_1 \blacktriangleright \theta_1 \;\text{ and }\; I \Vdash_\Sigma G_2, \Delta' \blacktriangleright \mathcal{C}_2 \blacktriangleright \theta_2,$$

$\theta_{|FV(G_1,\Delta')} = (\theta_1 \circ \sigma_1)_{|FV(G_1,\Delta')}$, $\mathcal{C}_1\theta_1\sigma_1 \preccurlyeq \mathcal{C}$, $\theta_{|FV(G_2,\Delta')} = (\theta_2 \circ \sigma_2)_{|FV(G_2,\Delta')}$ and $\mathcal{C}_2\theta_2\sigma_2 \preccurlyeq \mathcal{C}$.

Now, let $\mathcal{D}_1 \preccurlyeq \mathcal{C}_1$ and $\mathcal{D}_2 \preccurlyeq \mathcal{C}_2$ s.t. $\mathcal{D}_1\theta_1\sigma_1 = \mathcal{D}_2\theta_2\sigma_2 = \mathcal{C}_1\theta_1\sigma_1 \cap \mathcal{C}_2\theta_2\sigma_2$. Let τ be the substitution $(\theta_1 \circ \sigma_1)_{|FV(G_1,\Delta',\mathcal{C}_1)} \cup (\theta_2 \circ \sigma_2)_{|FV(G_2,\Delta',\mathcal{C}_2)}$; τ is well defined because $\theta_1 \circ \sigma_1$ and $\theta_2 \circ \sigma_2$ both behave like θ on variables in $FV(G_1, \Delta') \cap FV(G_2, \Delta')$, and $\mathcal{C}_1, \mathcal{C}_2$ do not have variables in common except for variables in G_1, G_2, Δ' (note that new variants of elements in I are chosen every time the judgment $\Vdash_\Sigma$ is computed).

Now, $\mathcal{D}_1$ and $\mathcal{D}_2$ are unified by τ, because $\mathcal{D}_1\tau = \mathcal{D}_1\theta_1\sigma_1 = \mathcal{D}_2\theta_2\sigma_2 = \mathcal{D}_2\tau$. Therefore, there exists $\theta_3 = m.g.u.(\mathcal{D}_1, \mathcal{D}_2)$ s.t. $\tau \geq \theta_3$ (θ_3 is more general than τ). Also, $\tau \geq \theta_1\sigma_1 \geq \theta_1$ and $\tau \geq \theta_2\sigma_2 \geq \theta_2$. Therefore, τ is an upper bound for $\{\theta_1, \theta_2, \theta_3\}$, hence there exist $\theta' = (\theta_1 \uparrow \theta_2 \uparrow \theta_3)_{|FV(G_1,G_2,\Delta',\mathcal{C})}$, and a substitution γ s.t. $\tau = \theta' \circ \gamma$. Now we can apply $\Vdash_\Sigma$ definition (rule for $\&$) and we get that

$$I \Vdash_\Sigma G_1 \,\&\, G_2, \Delta' \blacktriangleright \mathcal{C}' \blacktriangleright \theta',$$

where $\mathcal{C}' = \mathcal{C}_1 + (\mathcal{C}_2 \setminus \mathcal{D}_2)$. Let $\sigma = \gamma$, and let's prove the thesis.

First of all, since $\theta' \circ \sigma = \theta' \circ \gamma = \tau$, and by definition of τ, we have that $\theta_{|FV(G_1,G_2,\Delta')} = (\theta' \circ \sigma)_{|FV(G_1,G_2,\Delta')}$. It remains to prove that $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}$. Now, $\mathcal{C}'\theta'\sigma = \mathcal{C}'\tau = \mathcal{C}_1\tau + \mathcal{C}_2\tau \setminus \mathcal{D}_2\tau = \mathcal{C}_1\tau + \mathcal{C}_2\tau \setminus \mathcal{D}_2\theta_2\sigma_2 = \mathcal{C}_1\tau + \mathcal{C}_2\tau \setminus (\mathcal{C}_1\theta_1\sigma_1 \cap \mathcal{C}_2\theta_2\sigma_2)$ $= \mathcal{C}_1\tau + \mathcal{C}_2\tau \setminus (\mathcal{C}_1\tau \cap \mathcal{C}_2\tau) \preccurlyeq \mathcal{C}$. The last passage holds because $\mathcal{C}_1\tau \preccurlyeq \mathcal{C}$ and $\mathcal{C}_2\tau \preccurlyeq \mathcal{C}$ (by definition of τ and by inductive hypothesis) and relies on the following property of multisets: $\mathcal{A} \preccurlyeq \mathcal{D}$ and $\mathcal{B} \preccurlyeq \mathcal{D}$ implies $\mathcal{A} + \mathcal{B} \setminus (\mathcal{A} \cap \mathcal{B}) \preccurlyeq \mathcal{D}$;

- if $\Delta = G_1 \parr G_2, \Delta'$ of $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

□

The satisfiability judgment $\Vdash_\Sigma$ also satisfies the following properties.

Lemma 10.30 *For any interpretations I_1, I_2, ..., context Δ, fact $\mathcal{C}$, and substitution θ,*

i. if $I_1 \sqsubseteq I_2$ and $I_1 \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ then there exist a fact $\mathcal{C}'$, and substitutions θ' and σ s.t. $I_2 \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(\Delta)} = (\theta' \circ \sigma)_{|FV(\Delta)}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}\theta$;

ii. if $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ and $\bigsqcup_{i=1}^{\infty} I_i \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ then there exist $k \in \mathbb{N}$, a fact $\mathcal{C}'$, and substitutions θ' and σ s.t. $I_k \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(\Delta)} = (\theta' \circ \sigma)_{|FV(\Delta)}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}\theta$.

Proof

i. Suppose $I_1 \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ and $I_1 \sqsubseteq I_2$. By Lemma 10.29 *i*, $[\![I_1]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}\theta$. By Lemma 10.9 *i*, $[\![I_2]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}\theta$. The conclusion then follows from Lemma 10.29 *ii*;

ii. Suppose $\bigsqcup_{i=1}^{\infty} I_i \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ and $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$. By Lemma 10.29 *i*, $[\![\bigsqcup_{i=1}^{\infty} I_i]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}\theta$, i.e., as it can be readily verified from Definition 10.21 and Definition 10.23, $\bigcup_{i=1}^{\infty} [\![I_i]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}\theta$. By Lemma 10.9 *ii*, there exists $k \in \mathbb{N}$ s.t. $[\![I_k]\!] \models_\Sigma \Delta\theta \blacktriangleright \mathcal{C}\theta$. The conclusion then follows from Lemma 10.29 *ii*.

□

We are now ready to define the abstract fixpoint operator $S_P : \mathcal{I} \to \mathcal{I}$. As usual, we will proceed in two steps. We will first define an operator working over interpretations (i.e., elements of $\mathcal{P}(HB(P))$). With a little bit of overloading, we will call the operator with the same name, i.e., S_P. This operator should satisfy the equation $[\![S_P(I)]\!] = T_P([\![I]\!])$ for every interpretation I. This property ensures soundness and completeness of the *symbolic* representation.

After defining the operator over $\mathcal{P}(HB(P))$, we will lift it to our abstract domain $\mathcal{I}$ consisting of the equivalence classes of elements of $\mathcal{P}(HB(P))$ w.r.t. the relation $\simeq$ defined in Definition 10.21. Formally, we first introduce the following definition.

Definition 10.31 (Symbolic Fixpoint Operator S_P) *Given an LO$_\forall$ program P and an interpretation I, the symbolic fixpoint operator S_P is defined as follows:*

$$S_P(I) = \{(\widehat{H} + \mathcal{C})\,\theta \mid (H \circ\!\!- G) \in \mathit{Vrn}(P),\ I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta\}.$$

Note that the S_P operator is defined using the judgment $\Vdash_{\Sigma_P}$.

Proposition 10.33 states that S_P is sound and complete w.r.t T_P. In order to prove it, we need to formulate Lemma 10.32 below.

Notation. Let P be an LO$_\forall$ program, and $\Sigma, \Sigma_1 \in \mathit{Sig}_P$ be two signatures such that $\Sigma_1 \subseteq \Sigma$. Given a fact $\mathcal{C}$, defined on Σ, we use $\lceil \mathcal{C} \rceil_{\Sigma \to \Sigma_1}$ to denote any fact which is obtained in the following way. For every constant (eigenvariable) $c \in (\Sigma \setminus \Sigma_1)$, pick a new variable in $\mathcal{V}$ (not appearing in $\mathcal{C}$), let it be x_c (distinct variables must be chosen for distinct eigenvariables). Now, $\lceil \mathcal{C} \rceil_{\Sigma \to \Sigma_1}$ is obtained by $\mathcal{C}$ by replacing every $c \in (\Sigma \setminus \Sigma_1)$ with x_c. For instance, if $\mathcal{C} = \{p(x, f(c)), q(y, d)\}$, with $c \in (\Sigma \setminus \Sigma_1)$ and $d \in \Sigma_1$, we have that $\lceil \mathcal{C} \rceil_{\Sigma \to \Sigma_1} = \{p(x, f(x_c)), q(y, d)\}$.

Given a context (multiset of goals) Δ, defined on Σ, we define $\lceil \Delta \rceil_{\Sigma \to \Sigma_1}$ in the same way. Similarly, given a substitution θ, defined on Σ, we use the notation $\lceil \theta \rceil_{\Sigma \to \Sigma_1}$ to denote the substitution obtained from θ by replacing every $c \in (\Sigma \setminus \Sigma_1)$ with a new variable x_c in

every binding of θ. For instance, if $\theta = [u \mapsto p(x, f(c)), v \mapsto q(y, d)]$, with $c \in (\Sigma \setminus \Sigma_1)$ and $d \in \Sigma_1$, we have that $\lceil\theta\rceil_{\Sigma\to\Sigma_1} = [u \mapsto p(x, f(x_c)), v \mapsto q(y, d)]$.

Using the notation $[\![I]\!] \models_{\Sigma_1} \lceil\Delta\rceil_{\Sigma\to\Sigma_1} \blacktriangleright \lceil\mathcal{C}\rceil_{\Sigma\to\Sigma_1}$ we mean the judgment obtained by replacing every $c \in (\Sigma \setminus \Sigma_1)$ with x_c simultaneously in $\mathcal{D}$ and $\mathcal{C}$. Newly introduced variables must not appear in Δ, $\mathcal{C}$, or I.

When Σ and Σ_1 are clear from the context, we simply write $\lceil\mathcal{C}\rceil$, $\lceil\Delta\rceil$, and $\lceil\theta\rceil$ for $\lceil\mathcal{C}\rceil_{\Sigma\to\Sigma_1}$, $\lceil\Delta\rceil_{\Sigma\to\Sigma_1}$, and $\lceil\theta\rceil_{\Sigma\to\Sigma_1}$.

Finally, we use $\xi_{\Sigma_1\to\Sigma}$ (or simply ξ if it is not ambiguous) to denote the substitution which maps every variable x_c back to c (for every $c \in (\Sigma \setminus \Sigma_1)$), i.e., consisting of all bindings of the form $x_c \mapsto c$ for every $c \in \Sigma \setminus \Sigma_1$. Clearly, we have that $\lceil F\rceil\xi = F$, for any fact or context F, and $\lceil\theta\rceil \circ \xi = \theta$ for any substitution θ.

Note that, by definition, $\lceil\mathcal{C}\rceil_{\Sigma\to\Sigma_1}$ and $\lceil\Delta\rceil_{\Sigma\to\Sigma_1}$ are defined on Σ_1, while $\xi_{\Sigma_1\to\Sigma}$ is defined on Σ.

Lemma 10.32 *Let P be an $LO_\forall$ program, I an interpretation, and $\Sigma, \Sigma_1 \in Sig_P$ two signatures, with $\Sigma_1 \subseteq \Sigma$.*

i. If $I \Vdash_{\Sigma_1} \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ then $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$;

ii. If $[\![I]\!] \models_\Sigma \Delta \blacktriangleright \mathcal{C}$ then $[\![I]\!] \models_{\Sigma_1} \lceil\Delta\rceil_{\Sigma\to\Sigma_1} \blacktriangleright \lceil\mathcal{C}\rceil_{\Sigma\to\Sigma_1}$.

Proof

i. By simple induction on the derivation of $I \Vdash_{\Sigma_1} \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$.

ii. By induction on the derivation of $[\![I]\!] \models_\Sigma \Delta \blacktriangleright \mathcal{C}$.

- If $[\![I]\!] \models_\Sigma \top, \Delta \blacktriangleright \mathcal{C}$, immediate;
- suppose $[\![I]\!] \models_\Sigma \mathcal{A} \blacktriangleright \mathcal{C}$ and $\mathcal{A} + \mathcal{C} \in [\![I]\!]_\Sigma$. It follows that there exist $\mathcal{B} \in I$, a fact $\mathcal{D}$ and a substitution θ (defined on Σ) such that $\mathcal{A} + \mathcal{C} = \mathcal{B}\theta + \mathcal{D}$. Note that $\mathcal{B}$ is defined on Σ_P by definition of (abstract) interpretation.

  Now, $\lceil\mathcal{A}\rceil + \lceil\mathcal{C}\rceil = \lceil\mathcal{A} + \mathcal{C}\rceil = \lceil\mathcal{B}\theta + \mathcal{D}\rceil = \lceil\mathcal{B}\theta\rceil + \lceil\mathcal{D}\rceil =$ (remember that $\mathcal{B}$ is defined on $\Sigma_P \subseteq \Sigma_1$) $\mathcal{B}\lceil\theta\rceil + \lceil\mathcal{D}\rceil$. We can conclude that $\lceil\mathcal{A}\rceil + \lceil\mathcal{C}\rceil \in [\![I]\!]_{\Sigma_1}$ (note that $\mathcal{B} \in I$ and $\lceil\theta\rceil$, $\lceil\mathcal{D}\rceil$ are defined on Σ_1), it follows $[\![I]\!] \models_{\Sigma_1} \lceil\mathcal{A}\rceil \blacktriangleright \lceil\mathcal{C}\rceil$;
- suppose $[\![I]\!] \models_\Sigma \forall x.G, \Delta \blacktriangleright \mathcal{C}$ and $[\![I]\!] \models_{\Sigma,c} G[c/x], \Delta \blacktriangleright \mathcal{C}$, with $c \notin \Sigma$. From $\Sigma_1 \subseteq \Sigma$ we get $\Sigma_1, c \subseteq \Sigma, c$, therefore we can apply the inductive hypothesis.

  It follows that $[\![I]\!] \models_{\Sigma_1,c} \lceil G[c/x], \Delta\rceil \blacktriangleright \lceil\mathcal{C}\rceil$ iff $[\![I]\!] \models_{\Sigma_1,c} \lceil G[c/x]\rceil, \lceil\Delta\rceil \blacktriangleright \lceil\mathcal{C}\rceil$ iff (remember that $c \notin \Sigma\setminus\Sigma_1$ because $c \notin \Sigma$) $[\![I]\!] \models_{\Sigma_1,c} \lceil G\rceil[c/x], \lceil\Delta\rceil \blacktriangleright \lceil\mathcal{C}\rceil$. By $\models$ definition (remember that $c \notin \Sigma$ implies $c \notin \Sigma_1$), we get $[\![I]\!] \models_{\Sigma_1} \forall x.\lceil G\rceil, \lceil\Delta\rceil \blacktriangleright \lceil\mathcal{C}\rceil$

iff $[\![I]\!] \models_{\Sigma_1} \lceil \forall x.G, \Delta \rceil \blacktriangleright \lceil \mathcal{C} \rceil$ (we assume x to be disjoint with the variables introduced by the $\lceil \cdot \rceil$ construction);

- the remaining cases follow by a straightforward application of the inductive hypothesis.

□

Proposition 10.33
For every $LO_\forall$ program P and interpretation I, $[\![S_P(I)]\!] = T_P([\![I]\!])$.

Proof

- $[\![S_P(I)]\!] \subseteq T_P([\![I]\!])$.

We prove that for every $\Sigma \in Sig_P$, $[\![S_P(I)]\!]_\Sigma \subseteq T_P([\![I]\!])_\Sigma$. Suppose $(\widehat{H} + \mathcal{C})\theta \in S_P(I)$, with $H \circ\!\!- G$ variant of a clause in P and $I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta$. Suppose also that $\mathcal{A} = ((\widehat{H} + \mathcal{C})\theta + \mathcal{D})\theta' \in Inst_\Sigma(Up_\Sigma(S_P(I))) = [\![Sp(I)]\!]_\Sigma$.

We have that $I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta$ implies $I \Vdash_\Sigma G \blacktriangleright \mathcal{C} \blacktriangleright \theta$ by Lemma 10.32 *i* (remember that $\Sigma_P \subseteq \Sigma$). Therefore, by Lemma 10.29 *i*, we get $[\![I]\!] \models_\Sigma G\theta\theta' \blacktriangleright \mathcal{C}'\theta'$ for any fact $\mathcal{C}' \succcurlyeq \mathcal{C}\theta$. Taking $\mathcal{C}' = \mathcal{C}\theta + \mathcal{D}$, it follows that $[\![I]\!] \models_\Sigma G\theta\theta' \blacktriangleright \mathcal{C}\theta\theta' + \mathcal{D}\theta'$. Therefore, by T_P definition, we have $\widehat{H}\theta\theta' + \mathcal{C}\theta\theta' + \mathcal{D}\theta' \in (T_P([\![I]\!]))_\Sigma$, i.e., $\mathcal{A} \in (T_P([\![I]\!]))_\Sigma$.

- $T_P([\![I]\!]) \subseteq [\![S_P(I)]\!]$.

We prove that for every $\Sigma \in Sig_P$, $Tp([\![I]\!])_\Sigma \subseteq [\![S_P(I)]\!]_\Sigma$. Suppose $\mathcal{A} \in (T_P([\![I]\!]))_\Sigma$. By definition of T_P, there exist a variant of a clause $H \circ\!\!- G$ in P, a fact $\mathcal{C}$ and a substitution θ (defined over Σ) s.t. $\mathcal{A} = \widehat{H}\theta + \mathcal{C}$ and $[\![I]\!] \models_\Sigma G\theta \blacktriangleright \mathcal{C}$.

By Lemma 10.32 *ii* we have that $[\![I]\!] \models_\Sigma G\theta \blacktriangleright \mathcal{C}$ implies $[\![I]\!] \models_{\Sigma_P} \lceil G\theta \rceil \blacktriangleright \lceil \mathcal{C} \rceil$ (hereafter, we use the notation $\lceil \cdot \rceil$ for $\lceil \cdot \rceil_{\Sigma \to \Sigma_P}$). From $H \circ\!\!- G$ in P, we know that G is defined on Σ_P. It follows easily that $\lceil G\theta \rceil = G\lceil \theta \rceil$, so that $[\![I]\!] \models_{\Sigma_P} G\lceil \theta \rceil \blacktriangleright \lceil \mathcal{C} \rceil$. By Lemma 10.29 *ii*, there exist a fact $\mathcal{C}'$, and substitutions θ' and σ (defined over Σ_P) s.t. $I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\lceil \theta \rceil_{|FV(G)} = (\theta' \circ \sigma)_{|FV(G)}$, and $\mathcal{C}'\theta'\sigma \preccurlyeq \lceil \mathcal{C} \rceil$.

By S_P definition, we have $(\widehat{H} + \mathcal{C}')\theta' \in S_P(I)$.

Now, $\mathcal{A} = \widehat{H}\theta + \mathcal{C} = \widehat{H}\lceil \theta \rceil \xi + \lceil \mathcal{C} \rceil \xi =$ (note that by hypothesis $\theta' \circ \sigma$ and $\lceil \theta \rceil$ coincide for variables in G, and are not defined on variables in H which do not appear in G because $H \circ\!\!- G$ is a variant) $\widehat{H}\theta'\sigma\xi + \lceil \mathcal{C} \rceil \xi \succcurlyeq \widehat{H}\theta'\sigma\xi + \mathcal{C}'\theta'\sigma\xi = ((\widehat{H} + \mathcal{C}')\theta')\sigma\xi \in [\![(\widehat{H} + \mathcal{C}')\theta']\!]_\Sigma \subseteq [\![S_P(I)]\!]_\Sigma$. □

The following corollary holds.

Corollary 10.34 *For every $LO_\forall$ program P and interpretations I and J, if $I \simeq J$ then $S_P(I) \simeq S_P(J)$.*

Proof If $I \simeq J$, i.e., $[\![I]\!] = [\![J]\!]$, we have that $T_P([\![I]\!]) = T_P([\![J]\!])$. By Proposition 10.33, it follows that $[\![S_P(I)]\!] = [\![S_P(J)]\!]$, i.e., $S_P(I) \simeq S_P(J)$. □

The previous Corollary allows us to safely lift S_P definition from the lattice $\langle \mathcal{P}(HB(P)), \subseteq \rangle$ to $\langle \mathcal{I}, \sqsubseteq \rangle$. Formally, we define the abstract fixpoint operator as follows.

Definition 10.35 (Abstract Fixpoint Operator S_P) *Given an $LO_\forall$ program P and an equivalence class $[I]_\simeq$ of $\mathcal{I}$, the abstract fixpoint operator S_P is defined as follows:*

$$S_P([I]_\simeq) = [S_P(I)]_\simeq$$

where $S_P(I)$ is defined in Definition 10.31.

For the sake of simplicity, in the following we will often use I to denote its class $[I]_\simeq$, and we will simply use the term *(abstract) interpretation* to refer to an equivalence class, i.e., an element of $\mathcal{I}$. The abstract fixpoint operator S_P satisfies the following property.

Proposition 10.36 (Monotonicity and Continuity) *For every $LO_\forall$ program P, the abstract fixpoint operator S_P is monotonic and continuous over the lattice $\langle \mathcal{I}, \sqsubseteq \rangle$.*

Proof

Monotonicity.

We prove that if $I \sqsubseteq J$, then $S_P(I) \sqsubseteq S_P(J)$, i.e., $[\![S_P(I)]\!] \subseteq [\![S_P(J)]\!]$. To prove this latter condition, we will use the characterization given by Proposition 10.24. Suppose $\mathcal{A} = (\widehat{H} + \mathcal{C})\theta \in S_P(I)$, with $H \circ\!\!- G$ variant of a clause in P and $I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta$.

By Lemma 10.30 *i*, there exist a fact $\mathcal{C}'$, and substitutions θ' and σ (note that they are defined over Σ_P) s.t. $J \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(G)} = (\theta' \circ \sigma)_{|FV(G)}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}\theta$. Let $\mathcal{C}\theta = \mathcal{C}'\theta'\sigma + \mathcal{D}$, with $\mathcal{D}$ a fact defined over Σ_P. By S_P definition, $\mathcal{B} = (\widehat{H} + \mathcal{C}')\theta' \in S_P(J)$.

Now, $\mathcal{A} = (\widehat{H} + \mathcal{C})\theta = \widehat{H}\theta + \mathcal{C}\theta = \widehat{H}\theta'\sigma + \mathcal{C}'\theta'\sigma + \mathcal{D}$ (note in fact that by hypothesis $\theta'\sigma$ and θ coincide for variables in G, and are not defined on variables in H which do not appear in G because $H \circ\!\!- G$ is a variant). Therefore, we have that $\mathcal{A} = \widehat{H}\theta'\sigma + \mathcal{C}'\theta'\sigma + \mathcal{D} = \mathcal{B}\sigma + \mathcal{D}$.

Continuity.

We show that S_P is finitary, i.e., if $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, then $S_P(\bigsqcup_{i=1}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=1}^{\infty} S_P(I_i)$, i.e.,

$[\![S_P(\bigsqcup_{i=1}^{\infty} I_i)]\!] \subseteq [\![\bigsqcup_{i=1}^{\infty} S_P(I_i)]\!]$. Again, we will use the characterization given by Proposition 10.24. Suppose $\mathcal{A} = (\widehat{H} + \mathcal{C})\theta \in S_P(\bigsqcup_{i=1}^{\infty} I_i)$, with $H \circ\!\!- G$ variant of a clause in P and $\bigsqcup_{i=1}^{\infty} I_1 \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta$.

By Lemma 10.30 *ii*, there exist $k \in \mathbb{N}$, a fact $\mathcal{C}'$, and substitutions θ' and σ (note that they are defined over Σ_P) s.t. $I_k \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C}' \blacktriangleright \theta'$, $\theta_{|FV(G)} = (\theta' \circ \sigma)_{|FV(G)}$, $\mathcal{C}'\theta'\sigma \preccurlyeq \mathcal{C}\theta$. Let $\mathcal{C}\theta = \mathcal{C}'\theta'\sigma + \mathcal{D}$, with $\mathcal{D}$ a fact defined over Σ_P. By S_P definition, $\mathcal{B} = (\widehat{H} + \mathcal{C}')\theta' \in S_P(I_k)$.

Exactly as above, we prove that $\mathcal{A} = (\widehat{H} + \mathcal{C})\theta = \widehat{H}\theta'\sigma + \mathcal{C}'\theta'\sigma + \mathcal{D} = \mathcal{B}\sigma + \mathcal{D}$. □

Corollary 10.37 *For every $LO_\forall$ program P, $[\![lfp(S_P)]\!] = lfp(T_P)$.*

Let $\mathcal{F}_{sym}(P) = lfp(S_P)$, then we have the following main theorem.

Theorem 10.38 (Soundness and Completeness) *For every $LO_\forall$ program P, $O(P) = F(P) = [\![\mathcal{F}_{sym}(P)]\!]_{\Sigma_P}$.*

Proof From Theorem 10.14 and Corollary 10.37. □

The previous results give us an algorithm to compute the operational and fixpoint semantics of a program P via the fixpoint operator S_P.

Example 10.39 Let us consider a signature with a constant symbol a, a function symbol f and predicate symbols p, q, r, s. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$. Let us consider the program P of Example 3.15, which is given below.

1. $r(w) \circ\!\!- q(f(w))$
2. $s(z) \circ\!\!- \forall x.p(f(x))$
3. $\bot \circ\!\!- q(u) \,\&\, r(v)$
4. $p(x) \parr q(x) \circ\!\!- \top$

From clause 4, and using the first rule for $\Vdash_{\Sigma_P}$, we get that $S_P(\emptyset) = [\{\{p(x), q(x)\}\}]_{\simeq}$. For simplicity, we omit the class notation, and we write

$$S_P{\uparrow_1} = S_P(\emptyset) = \{\{p(x), q(x)\}\}.$$

We can now apply the remaining clauses to the element $I = \{p(x), q(x)\}$ (remember that $S_P([I]_{\simeq}) = [S_P(I)]_{\simeq}$). From the first clause (see Example 10.27) we have $I \Vdash_{\Sigma_P} q(f(w')) \blacktriangleright p(x') \blacktriangleright [x' \mapsto f(w')]$. It follows $(r(w'), p(x'))[x' \mapsto f(w')] = r(w'), p(f(w')) \in S_P{\uparrow_2}$. As the reader can verify (see discussion in Example 10.27), clause 2 does not yield

any further element, and the same holds for clause 3, therefore we can assume (changing w' into y for convenience)

$$S_P{\uparrow}_2 = \{\{p(x), q(x)\}, \{r(y), p(f(y))\}\}.$$

Now, we can apply clause 3 to elements in $S_P{\uparrow}_2$. According to Example 10.27, we have that $I \Vdash_{\Sigma_P} q(u') \,\&\, r(v') \blacktriangleright p(x''') \blacktriangleright [u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')]$. Therefore we get that $(p(x'''))[u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')] = p(f(y')) \in S_P{\uparrow}_3$. Clause 2 cannot be applied yet, for the same reasons as above. Also, note that the element $r(y), p(f(y))$ is now subsumed by $p(f(y'))$. Therefore we can assume

$$S_P{\uparrow}_3 = \{\{p(x), q(x)\}, \{p(f(y'))\}\}.$$

Finally, we can apply clause 2 to $S_P{\uparrow}_3$, using the $\forall$-rule for the $\Vdash_{\Sigma_P}$ judgment. Take $c \notin \Sigma_P$, and consider a renaming of the last element in $S_P{\uparrow}_3$, $p(f(y''))$. Consider (a renaming of) clause 2, $s(z') \circ\!\!- \forall x.p(f(x))$. We have that $I \Vdash_{\Sigma_P, c} p(f(c)) \blacktriangleright \epsilon \blacktriangleright nil$, with nil being the empty substitution. Therefore we get that $I \Vdash_{\Sigma_P} \forall x.p(f(x)) \blacktriangleright \epsilon \blacktriangleright nil$, from which $s(z') \in S_P{\uparrow}_4$. The reader can verify that no further clauses can be applied and that $S_P{\uparrow}_4$ is indeed the fixpoint of S_P, therefore we have that

$$S_P{\uparrow}_4 = S_P{\uparrow}_\omega = \{\{p(x), q(x)\}, \{p(f(y'))\}, \{s(z')\}\}.$$

Note that $F(P)$ is defined to be $[\![lfp(S_P)]\!]_{\Sigma_P}$, therefore it includes, e.g., the elements $s(a)$ (see Example 3.15), $p(f(f(y')))$ and $p(f(f(y'))), q(x'')$. □

10.4 Ensuring Termination

In this section we will rephrase the results of Section 9.3, concerning termination of the bottom-up evaluation algorithm, to the case of $\mathrm{LO}_\forall$ programs. An application of these results will be presented in Section 10.5.

Accordingly, we introduce the class of programs with *monadic* predicate symbols.

Definition 10.40 (The $\mathcal{P}_1(\mathbf{LO}_\forall)$ class) *The class $\mathcal{P}_1(LO_\forall)$ consists of $LO_\forall$ programs built over a signature Σ including a* finite *set of constant symbols, no function symbols, and a* finite *set of predicate symbols with arity at most one.*

Definition 10.41 (Multisets $\mathcal{M}_1(\mathbf{LO}_\forall)$) *The class $\mathcal{M}_1(LO_\forall)$ consists of multisets of (non ground) atomic formulas over a signature Σ including a* finite *set of constant symbols, no function symbols, and predicate symbols with arity at most one.*

Example 10.42 Let Σ be a signature including a constant symbols a, no function symbols, and predicate symbols p, q (with arity one) and r (with arity zero). Let $\mathcal{V}$ be a denumerable set of variables, and $x, y, \ldots \in \mathcal{V}$. Then the clause

$$p(x) \parr q(x) \parr q(y) \parr r \circ\!\!- (p(a) \parr p(y)) \,\&\, \forall z.q(z)$$

is in the class $\mathcal{P}_1(LO_\forall)$, and the multiset $\{p(x), q(x), q(a), r\}$ is in the class $\mathcal{M}_1(LO_\forall)$. □

The following result holds.

Proposition 10.43 *The class $\mathcal{M}_1(LO_\forall)$ is closed under applications of S_P, i.e., if $I \subseteq \mathcal{M}_1(LO_\forall)$ then $S_P(I) \subseteq \mathcal{M}_1(LO_\forall)$.*

Proof Immediate by Definition 10.31 and Definition 10.26. □

Termination of the bottom-up evaluation for the class $\mathcal{P}_1(LO_\forall)$ is stated in the following proposition.

Proposition 10.44 *Let P be a $\mathcal{P}_1(LO_\forall)$ program. Then there exists $k \in \mathbb{N}$ such that $\mathcal{F}_{sym}(P) = \bigsqcup_{i=0}^{k} S_P{\uparrow_k}\,(\emptyset)$.*

Proof The proof is carried out, similarly to that of Proposition 9.57, by proving that the entailment relation between multisets in the class $\mathcal{M}_1(LO_\forall)$ (see Proposition 10.24) is a wqo.

Entailment between multisets in $\mathcal{M}_1(LO_\forall)$ being a wqo is a simple consequence of Proposition 9.57. Take a multiset in the class $\mathcal{M}_1(LO_\forall)$. First of all, perform the following transformation: for every atom $p(a)$, where a is a constant symbol in Σ (note that there no other ground terms other than constants in this class) introduce a new predicate symbol, with arity zero, let it be p_a, and transform the original multiset by substituting p_a in place of $p(a)$. The resulting set of predicate symbols is still finite (the set of constants of the program is finite).

Now, it is easy to see that entailment between multisets transformed in the above way is a *sufficient* condition for entailment of the original multisets (note that the condition is not *necessary*, e.g. I cannot recognize that $p(a)$ entails $p(x)$). Now, entailment between transformed multisets is a wqo. In fact, note that a transformed multiset consists of atomic formulas with a predicate symbol (with arity zero or one), where arguments are (possibly duplicated) variables. Now, duplicated variables correspond to *equality* constraints in the class $\mathcal{M}_1(\mathrm{NC})$ of Section 9.3.

Using a similar proof to that of Proposition 9.52, and from Propositions 10.43 and 6.23, we can conclude. □

10.5 An Example: A Distributed Test-and-Lock Protocol

We conclude this chapter by discussing an example, which can be seen as a specification for a distributed *test-and-lock* protocol for a net with multiple resources, each one controlled by a monitor. We have verified protocol correctness using the tool described in Appendix A.

The protocol is as follows. A set of *resources*, distinguished by means of *resource identifiers*, and an arbitrary set of processes are given. Processes can non-deterministically request access to any resource. Access to a given resource must be exclusive (only one process at a time). Mutual exclusion is enforced by providing each resource with a *semaphore*.

Given a propositional symbol *init*, we can encode the initial states of the system as follows (we intentionally introduce a flaw which we will disclose later):

1. $init \circ\!\!- init \parr think$
2. $init \circ\!\!- init \parr m(x, unlocked)$
3. $init \circ\!\!- \bot$

The atom *think* represents a thinking (idle) process, while the first-order atom $m(x, s)$ represents a *monitor* for the resource with identifier x and associated semaphore s. The semaphore s can assume one of the two values *locked* or *unlocked*. Clause 1 and clause 2 can modify the initial state by adding, respectively, an arbitrary number of thinking processes and an arbitrary number of resources (with an initially unlocked semaphore). Finally, using clause 3 the atom *init* can be removed after the initialization phase.

The core of the protocol works as follows:

4. $think \circ\!\!- wait(x)$
5. $wait(x) \circ\!\!- think$
6. $wait(x) \parr m(x, unlocked) \circ\!\!- use(x) \parr m(x, locked)$
7. $use(x) \parr m(x, locked) \circ\!\!- think \parr m(x, unlocked)$

Using clause 4, a process can non-deterministically request access to any resource with identifier x, moving to a waiting state represented by the atom $wait(x)$. Clause 5 allows

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash_\Sigma init, \top, m(x, locked), m(x, locked)}\ \top_r}{P \vdash_\Sigma init, use(x), use(x), m(x, locked), m(x, locked)}\ bc^{(8)}}{P \vdash_\Sigma init, wait(x), wait(x), m(x, unlocked), m(x, unlocked)}\ bc^{(6^*)}}{P \vdash_\Sigma init, think, think, m(x, unlocked), m(x, unlocked)}\ bc^{(4^*)}}{P \vdash_\Sigma init, think, think}\ bc^{(2^*)}}{P \vdash_\Sigma init}\ bc^{(1^*)}$$

Figure 10.2: A trace violating mutual exclusion for the test-and-lock protocol

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{P \vdash_{\Sigma,c,d} use(c), wait(d), think, m(c, locked), m(d, unlocked)}}{P \vdash_{\Sigma,c,d} wait(c), wait(d), think, m(c, unlocked), m(d, unlocked)}\ bc^{(6)}}{P \vdash_{\Sigma,c,d} wait(c), wait(d), use(c), m(c, locked), m(d, unlocked)}\ bc^{(7)}}{P \vdash_{\Sigma,c,d} wait(c), wait(d), wait(c), m(c, unlocked), m(d, unlocked)}\ bc^{(6)}}{P \vdash_{\Sigma,c,d} think, wait(d), wait(c), m(c, unlocked), m(d, unlocked)}\ bc^{(4)}}{P \vdash_{\Sigma,c,d} think, think, wait(c), m(c, unlocked), m(d, unlocked)}\ bc^{(4)}}{P \vdash_{\Sigma,c,d} think, think, think, m(c, unlocked), m(d, unlocked)}\ bc^{(4)}}{P \vdash_{\Sigma,c,d} init, think, think, think, m(c, unlocked), m(d, unlocked)}\ bc^{(3)}}{P \vdash_\Sigma init, think, think, think}\ bc^{(2'^*)}}{P \vdash_\Sigma init}\ bc^{(1^*)}$$

Figure 10.3: A test-and-lock protocol: example trace

$\{use(x),\ use(x)\}$
$\{m(x, unlocked),\ use(x),\ init\}$
$\{m(x, unlocked),\ use(x),\ wait(y)\}$
$\{m(x, unlocked),\ use(x),\ use(y),\ m(y, locked)\}$
$\{m(x, locked),\ use(x),\ m(y, unlocked),\ m(y, unlocked),\ think\}$
$\{m(x, unlocked),\ m(x, unlocked),\ wait(y),\ think\}$
$\{m(x, unlocked),\ m(x, unlocked),\ use(y),\ m(y, locked),\ use(z),\ m(z, locked)\}$
$\{m(x, unlocked),\ m(x, unlocked),\ use(y),\ m(y, locked),\ wait(z)\}$
$\{wait(x),\ m(y, unlocked),\ m(y, unlocked),\ wait(z)\}$
$\{m(x, unlocked),\ m(x, unlocked),\ init\}$
$\{m(x, unlocked),\ m(x, unlocked),\ think,\ think\}$
$\{use(x),\ m(x, unlocked),\ think\}$

Figure 10.4: Fixpoint computed for the test-and-lock protocol

a process to go back to thinking from a waiting state. By clause 6, a waiting process can synchronize with the relevant monitor and is granted access provided the corresponding semaphore is unlocked. As a result, the semaphore is locked. The atom $use(x)$ represents a process which is currently using the resource with identifier x. Clause 7 allows a process to release a resource and go back to thinking, unlocking the corresponding semaphore.

Using our verification tool (see Appendix A), we can now automatically verify the *mutual exclusion* property for the above test-and-lock protocol. The specification of unsafe states is simply as follows:

$$8.\ \ use(x) \parr use(x) \circ\!\!- \top$$

The test-and-lock specification can be seen to fall into the class $\mathcal{P}_1(LO_\forall)$. In fact, the binary predicate symbol m plays the same role as two unary predicate symbols m_{locked} and $m_{unlocked}$ (note that the second argument of m in the initial specification is always instantiated). Termination of the fixpoint computation is therefore *guaranteed* by Proposition 10.44. Running the verification algorithm, we actually find a security violation. The corresponding trace is shown in Figure 10.2. The problem of the above specification lies in clause 2. In fact, using an (externally quantified) variable x does not prevent the creation of multiple monitors for the same resource. This causes a violation of mutual exclusion when different processes are allowed to concurrently access a given resource by different monitors.

Luckily, we can fix the above problem in a very simple way. As we do not care about what resource identifiers actually are, we can elegantly encode them using universal quantifica-

$\{use(x)\ , use(x)\}$
$\{m(x,y)\ , m(x,z)\}$
$\{m(x,unlocked)\ , use(x)\ , use(y)\ , m(y,z)\}$
$\{m(x,unlocked)\ , use(x)\ , wait(y)\}$
$\{m(x,unlocked)\ , use(x)\ , init\}$
$\{use(x)\ , m(x,unlocked)\ , think\}$

Figure 10.5: Fixpoint computed using invariant strengthening for the test-and-lock protocol

tion in the body of clause 2, as follows:

$$2'.\ \ init \multimap init \parr \forall x.m(x, unlocked)$$

Every time a resource is created, a new constant, acting as the corresponding identifier, is created as well. Note that by the operational semantics of universal quantification, *different* resources are assigned *different* identifiers. This clearly prevents the creation of multiple monitors for the same resource. An example trace for the modified specification is shown in Figure 10.3 (where P is the program consisting of clauses 1, 2', 3 through 8 above, and we followed the usual conventions, with $bc^{(i^*)}$ denoting multiple applications of clause number i).

Now, running again our verification tool on the corrected specification (termination is still guaranteed by Proposition 10.44), with the same set of unsafe states, we get the fixpoint shown in Figure 10.4 (where, for readability, we re-use the same variables in different multisets). The fixpoint contains 12 elements and is reached in 7 steps. As the fixpoint does not contain *init*, mutual exclusion is verified, *for any number of processes and any number of resources*.

We conclude by showing how it is possible to optimize the fixpoint computation by invariant strengthening (see Section 6.1.2.1). One possibility might be to apply the so-called *counting abstraction*, i.e., turn the above LO specification into a propositional program (i.e., a Petri net) by abstracting first-order atoms into propositional symbols (e.g. $wait(x)$ into $wait$, and so on), and compute the structural invariants of the corresponding Petri net. However, this strategy is not helpful in this case (no meaningful invariant is found). We can still try some invariants using some *ingenuity*. For instance, consider the following invariant:

$$9.\ \ m(x,y) \parr m(x,z) \multimap \top$$

For what we said previously (*different* resources are assigned *different* identifiers) this invariant must hold for our specification. Running the verification tool on this extended specification we get the fixpoint in Figure 10.5, containing only 6 elements and converging in 4 steps. A further optimization could be obtained by adding the invariant $use(x) \parr m(x, unlocked) \multimap \top$ (intuitively, if someone is using a given resource, the corresponding

semaphore cannot be unlocked). In this case the computation converges immediately at the first step.

10.6 Related Work

In this section we report on some related work, in particular for what concerns automated deduction techniques for verification of parameterized systems. Alternative approaches based, e.g., on model checking, have been discussed in Section 9.5. Clearly, there may be overlaps between the two approaches. For instance, [FO97] uses an approach based on *regular sets* (see Section 9.5) in a deductive setting. Specifically, a methodology is proposed, which is able to infer, using a generalization method, a regular language to represent reachable states for parameterized rings.

Many methods proposed for the verification of parameterized systems, can be roughly classified into those using *explicit induction*, *network invariants*, which can be viewed as *implicit induction*, and *abstraction*. Among those using *explicit induction*, there are, e.g., [EN95, GS92]. In [EN95], for instance, it is shown that some properties of rings of arbitrary size can be verified by checking they hold for a *finite* number of instances of the problem. The resulting instances can then be verified by model checking techniques.

Implicit induction is used, e.g., in [WL89, LHR97, KM95]. A general structural induction theorem for concurrent models of processes is discussed in [KM95]. The induction method is based on representing a given specification as a process, and in finding a suitable invariant process to apply induction. In [LHR97] the problem of verification for linear networks of processes is reduced to the construction of a network invariant, which can be synthetized using heuristics, e.g. widening techniques [CC77].

Abstraction techniques are used, e.g., in [ID96, LS97, GZ98]. For instance, in [ID96] systems with replicated components are verified by explicit enumeration in an abstract state space in which states do not record the exact number of components. The notion of *conservative abstraction* is discussed in [LGS+95]. Namely, the paper discusses property preserving transformations, which ensure that a property for a given system can be safely verified on a simpler abstraction. The formal results are based on the theory of abstract interpretation [CC77]. An approach mixing abstraction and induction is the one given in [MQS00].

Finally, approaches which are more deductive in nature include for instance [RV95]. An application of this system to specification and verification of security protocols is given in [JRV00] (see Section 11.6). Traditional automated deduction approaches include for instance using the PVS system [ORR+96], which combines (interactive) proof checking with model checking techniques, and provides a meta-language for defining proof strategies. Other deductive tools include, e.g., LCF [GMW79], Nuprl [CAB+86],and HOL [GM93].

Summary of the Chapter. *In this chapter we have discussed a bottom-up semantics for a first-order formulation of LO, where universal quantification is used as a primitive for generating new values. We have shown that the usual ground semantics is not refined enough to capture the operational semantics of the universal quantifier. The semantics we have defined is the equivalent of the so-called non-ground success set semantics of Horn logic. The semantics is effective and complete with respect to the operational semantics, though in general evaluation can be non-terminating. As an example, we have verified that mutual exclusion holds for a parameterized version of a distributed test-and-lock protocol, where universal quantification is used to generate new resource identifiers.*

In the next chapter we will apply the methodology and techniques discussed in this chapter to the security protocol domain. We will encode security protocols using first-order LO specifications, and we will see how interesting properties like, e.g., confidentiality, can be automatically validated using our backward reachability strategy.

Chapter 11

A Case-Study: Security Protocols

In this chapter we will illustrate and apply the methodology for system verification discussed in this thesis to the security protocol domain. Given the enormous increase in the development and applications of networked and distributed systems which has taken place in recent years, the field of security protocols seems to be a particularly interesting and relevant test bed for any verification tool, as witnessed by the plentiful and varied literature on this subject. The design and implementation of security protocols are difficult and error-prone. As a meaningful example, we mention the case of Needham-Schroeder authentication protocol [NS78], which only recently (*seventeen* years after its publication) has been shown to be flawed [Low95]. That's why formal techniques to specify and analyze protocols have deserved great attention in the community.

In this chapter we will focus in particular on *authentication* protocols. Authentication protocols are a common means to synchronize and distribute information between agents (called *principals*) operating over a network. Typically, execution of a protocol run should provide the involved principals with some secret information, e.g. a cryptographic key, or knowledge about the other principals. Authentication protocols should be reliable enough to be used in a potentially compromised environment, so as to prevent a malicious intruder to cheat another agent about its own identity, or to get unauthorized information.

After presenting some background on security, in this chapter we will show how our verification tool can be used to specify, validate or detect *attacks* in security protocols. We will exemplify our methodology through a careful selection of different protocols presented in the literature on security. Formally, we will encode protocols in a fragment of first-order linear logic with universal quantification, corresponding to the language $\mathrm{LO}_\forall$ presented in Chapter 10. Our inspiration came from [CDL$^+$99], where an equivalent logical fragment, using *existential* quantification, is presented. Following [CDL$^+$99], we will use the universal quantifier to model the generation of new values (called *nonces*) which are commonly

used in the design of security protocols. A remarkable feature of our approach is that we can reason about *parametric* and *open* systems. As a result, we do not have to pose any artificial limitation on the number of parallel runs of a given protocol. In particular, we allow an agent to take part into different protocol sessions at the same time.

This chapter extends some preliminary results reported in [Boz01].

11.1 Introduction

In a distributed environment, it is necessary to provide the agents with a way to ensure each other's identity, and to exchange confidential information in a secure manner. This is the role of authentication protocols. Typically, running an authentication protocol provides the involved principals with information about the environment and the other principals, which can subsequently be used to take decisions on how to act. Clearly, it is crucial to ensure that these information may not be tampered with by a malicious intruder. In general, it is very difficult or even impossible to prevent every form of malicious *listening* over a network. The design of authentication protocols must take into account the possibility of messages sent over a network to be intercepted, and the presence of malicious agents which can impersonate the role of an honest participant in a given protocol run. Authentication protocols should be designed in such a way to be resistant to every form of *attack*, so as not to disclose private information to unauthorized agents, and to prevent an agent from cheating other agents about its own identity. Cryptographic primitives are a common means to achieve these goals, but they are not sufficient to ensure authentication.

In our view, formal verification of protocols is a process which is made up of two distinct, though related, phases: (1) choosing a suitable formalism to represent the environment and protocol execution (typically, in the form of a specification logic), and to specify security properties; (2) enriching the logic with a (hopefully complete and automatic) tool to formally verify that a given specification is correct with respect to a given property.

Neither of the two phases should be underestimated. In particular, phase (1) is responsible for fixing a suitable *abstraction* to represent protocol execution. Typically, simplifying assumptions on messages exchanged by agents and on cryptographic primitives are used in order to abstract from implementation-dependent details. For instance, according to the so-called *Dolev-Yao model* [DY83], messages are considered as indivisible abstract values, rather than sequences of bits, and encryption is modeled in an idealized way. Therefore, phase (2) alone is not sufficient to guarantee correctness. One should never forget that correctness results strongly rely on the abstraction chosen in phase (1), for instance using the Dolev-Yao model we will not be able to prove that a protocol is resistant to implementation-dependent attacks due to the particular crypto-algorithm used. Even most importantly, correctness results rely upon protocol specification given in phase (1) being faithful (in a

sense to be clarified) with respect to the given abstraction one has in mind.

Here, we present a comprehensive logical environment for the specification and analysis of protocols. Our position with respect to phase (1) mentioned above is to use a specification logic which is as close as possible to the usual informal presentation which can be found in the literature on security protocols. In particular, following [CDL+99] we will use a multiset-rewriting-like formalism to represent a given set of *principals* executing protocol sessions by exchanging messages over a network, with universal quantification providing a logical and clean way to express creation of *nonces*. As far as phase (2) is concerned, protocol verification will be carried out using the backward search strategy based on the bottom-up semantics for linear logic described in this thesis.

In particular, our approach is well suited to verify properties which can be specified by means of *minimality conditions* (e.g., a given state is unsafe if there are *at least* two principals which have completed the execution of a protocol and a given shared secret has been unintentionally disclosed to a third malicious agent). Our verification algorithm has connections both with (symbolic) model checking and with theorem proving. Unlike traditional approaches based on model-checking, we can reason about *parametric*, *infinite-state* systems, thus we do not pose any limitation on the number of parallel runs of a given protocol. We also allow a principal to take part into different sessions at the same time.

First of all, in the next section we give a brief overview about security and in particular about authentication protocols.

11.2 Some Background on Authentication

In this section we briefly discuss some background on authentication protocols and we fix some terminology and notations which will be used throughout the chapter.

Authentication protocols are used to coordinate the activity of different parties (e.g. users, hosts, or processes) operating over a network. These parties are usually referred to as **principals** in security literature. Execution of an authentication protocol can take place for various reasons. Principals can run an authentication protocol, e.g., to get some kind of information (e.g. cryptographic keys), or to make sure that another entity is *operational*, or to exchange *private* information with other principals using a secure communication channel. In general, a successful run of an authentication protocol will affect the subsequent decisions which the involved principals may take.

An authentication protocol generally consists of a *sequence* of messages exchanged between two or more principals (e.g. two users and a coordinating entity acting as a server). The form and number of exchanged messages is usually fixed in advance and must conform to a specific format. In general, a given principal can take part into a given protocol run in

different ways, e.g. as the **initiator** of the protocol or the **responder** (it is usually said that a principal can have different **roles**). Often, a principal is allowed to take part into different protocol runs simultaneously, possibly with different roles.

The design of authentication protocols must take into account the possibility of messages to be intercepted, and the presence of malicious agents which can impersonate the role of an honest principal. One of the key issues in authentication is to ensure **confidentiality**, i.e., to avoid private information being disclosed to unauthorized clients. Another issue is to prevent malicious principals from cheating by impersonating other principals. In general, a principal should have enough information to ensure that every message received has been created *recently* (as part of the current protocol run) and by the principal who claims to have sent it. Replaying of old messages should be detected. Authentication protocols must be designed in such a way to be resistant to every possible form of **attack**. In particular, interception of messages can prevent completion of a protocol run, but should never cause a leak of information or compromise security.

11.2.1 Cryptographic Prerequisites

Cryptographic primitives are a fundamental, though not sufficient, ingredient of authentication protocols. A message to be transmitted over a network is usually referred to as **plaintext** or **datagram**. The task of a cryptographic algorithm is to convert the given message to a form which is unintelligible to anyone else except the intended receiver. The conversion phase is called **encryption** and usually depends on an additional parameter known as **encryption key**, whereas the encoded message is referred to as **ciphertext** or **cryptogram**. The reverse phase of decoding is called **decryption**, and usually requires possession of the corresponding **decryption key**.

In **symmetric key cryptography**, the encryption key and the decryption key can be easily obtained from each other by public techniques (usually they are identical). Security of communication requires the keys to be kept secret between the relevant principals. The exact way encryption works depends on the particular crypto-algorithm used. Most algorithms work by encrypting *blocks* of plaintext at a time. The most famous of this class is the so-called Data Encryption Standard (**DES**) [DES76]. In **public key cryptography**, no secret is shared between communicating principals. Each principal A has a pair of keys, the first one being the *public* key, and the other the *private* key. The public key is made available and can be used to encrypt messages for principal A, whereas the private key is only known to A, who can use it to decrypt incoming messages. Some public key algorithm allow the private key to be used for encryption and the public key to be used for decryption. This mechanism is used to guarantee *authenticity* (rather than *confidentiality*) of messages. The most famous algorithm in this class was developed by Rivest, Shamir and Adleman, and is universally known as **RSA** [RSA78].

Authentication protocols (see [CJ97] for a survey) are usually classified depending on the cryptographic approach taken, e.g. symmetric key or public key protocols. Furthermore, a distinction is also made between protocols which use one or more **trusted third parties** (e.g. a central distribution key server) and protocols which do not.

11.2.2 A Classification of Attacks

Authentication protocols can be compromised by different forms of *attacks*. We present below a quick and rather informal classification (see also [CJ97]). Some examples of attacks of known protocols will be illustrated in Section 11.4.

Freshness Attacks: a freshness attack typically takes place when a principal is induced to accept, as part of the current protocol run, an old message which is currently being replayed by a malicious intruder. For instance, a principal can be led to accept an old and possibly compromised key as a new and legitimate one. Mechanisms for ensuring freshness of messages (e.g. *timestamps*) are often used to prevent such kind of attacks;

Type Flaw Attacks: a type flaw attack takes place when a principal is induced to erroneously interpret the structure of the current message. This may happen because, at the concrete level, a message is nothing but a flat sequence of bits. Mechanisms to prevent this kind of attacks usually rely on enriching messages with redundant information about their internal structure;

Parallel Session Attacks: a parallel session attack is usually carried out by a malicious intruder which forms messages for a given protocol run using messages coming another legitimate session which is executed concurrently;

Implementation Dependent Attacks: implementation dependent attacks are very subtle and can depend on a number of ways a given protocol is implemented. A typical area in which implementation dependent attacks can arise is given by the subtleties of the particular crypto-algorithm used and its interaction with specific protocols.

Other forms of attacks include, e.g., *binding attacks* and *encapsulation attacks*. We refer to [CJ97] for a discussion and several examples of all the different forms of attacks.

11.2.3 The Dolev-Yao Intruder Model

Most formal approaches to protocol specification and analysis, including ours, are based on a set of simplifying assumptions, which is known as the *Dolev-Yao* intruder model. This

model has been developed on the basis of some assumptions described by Needham and Schroeder in [NS78] and by Dolev and Yao in [DY83].

According to the so-called *Dolev-Yao model* [DY83], messages are considered as indivisible abstract values, instead of sequences of bits. Furthermore, the details of the particular crypto-algorithm used are abstracted away, giving rise to a *black-box* model of encryption (*perfect encryption*). This set of assumptions simplifies protocol analysis, although it has the drawback of preventing the discovery of implementation dependent attacks.

There seems to be no standard presentation of the Dolev-Yao intruder model. In general, it consists of a set *conservative* assumptions on the potentialities of any possible attacker. Typically, this model tries to depict a worst-case scenario, in which there is an intruder who has complete control of the network, so that he/she can intercept messages, block further transmission and/or replay them at any time, possibly modifying them. The intruder works by *decomposing* messages (provided he/she knows the key they are encrypted with), and *composing* new messages with the information in his/her possession. The intruder can also use an internal memory to store information, in general he/she knows the identity and, in the case of public-key encryption, the public keys of the other principals. The intruder is supposed *not* to know the *private* keys of the other principals, unless they have been disclosed in some way (clearly, without this latter assumption *any* protocol is breakable in general).

It has been proved that the Dolev-Yao intruder is, so to say, the *most powerful attacker* [Cer01a] for the given model under consideration (e.g. perfect encryption), in the sense that it can simulate the activity of any other possible attacker. Furthermore, in [SMC00] it has been proved that it is not restrictive to consider a single Dolev-Yao intruder instead of multiple ones.

11.2.4 An Informal Protocol Notation

In the literature on security, protocols are usually presented by means of an informal notation. In this section we explain this notation illustrating the so-called Needham-Schroeder public-key authentication protocol [NS78] (see also Section 11.4.2).

The protocol, in the usual notation, is as follows.

$$
\begin{array}{lrcl}
1. & A \rightarrow S & : & A, B \\
2. & S \rightarrow A & : & \{K_b, B\}_{K_s^{-1}} \\
3. & A \rightarrow B & : & \{N_a, A\}_{K_b} \\
4. & B \rightarrow S & : & B, A \\
5. & S \rightarrow B & : & \{K_a, A\}_{K_s^{-1}} \\
6. & B \rightarrow A & : & \{N_a, N_b\}_{K_a} \\
7. & A \rightarrow B & : & \{N_b\}_{K_b}
\end{array}
$$

The protocol is run to achieve authentication between two principals A and B. A central server S is in charge of distributing the public keys of principals. Messages 3, 6, and 7 are the core of the protocol, while the purpose of messages 1, 2, 4 and 5 is to get public keys from the central authority. The notation $\{M\}_K$ indicates a message with content M and encrypted with a key K. Also, by convention $A, B, \ldots$ indicate principal identifiers, K_a and K_b denote, respectively, A's and B's public keys, and K_s^{-1} is the private key of server S (encryption with the private key is used to ensure authenticity, see Section 11.2.1).

Now, the protocol has the following structure. A given principal A acts as *initiator* of the protocol, and asks the central authority for B's public key (message 1). The central authority sends back to A the required key (message 2: B's identity is included in the message to prevent attacks based on diverting key deliveries). Principal A creates a *nonce* (i.e., a newly generated value), called N_a, and sends it to B together with its own identity (message 3), encrypting the message with B's public key. Upon receiving this message, principal B decrypts the message, and in turn asks the central authority for the public key of A (message 4). After getting the server's reply (message 5), principal B generates a new nonce N_b, and sends both nonces, N_a and N_b, to A, encrypting the message with A's key (message 6). When A gets this message, a check is made that it contains the previously generated nonce N_a, and, if so, a new message, encrypted with B's key and including the last nonce N_b, is sent to B (message 7). The protocol is successfully completed provided B gets the previously generated nonce N_b.

Completion of the protocol should convince A about B's identity (and vice versa) and also provide A and B with two shared values (N_a and N_b) which they could use afterwards for authentication purposes. The use of **nonces** is ubiquitous in authentication protocols. Intuitively, a nonce should be considered as some sort of *random* and *unguessable* value. The purpose of nonces is to prevent a malicious intruder from attempting to break a given protocol by sending messages and pretending they have been generated by someone else. To exemplify, an attacker could possibly pretend to be principal B, intercept message 6 and replace it with a different message. However, the message created by the attacker will never be accepted as a legitimate message by A, unless it contains the nonce N_a. Unfortunately (for the attacker) nonce N_a is not known except to B (and A, of course), because only B can decrypt message 3.

Intuitively, assuming A and B behave honestly and their private keys are not known to anyone else, and assuming nonces are not guessable, the protocol should prevent a malicious intruder to impersonate one of the two principals. However, under certain conditions, this protocol fails to achieve authentication [Low95]. We will discuss this point in Section 11.4.2.

11.3 Specifying Authentication Protocols

In Section 11.4, we will illustrate the specification and analysis of different examples of protocols. Our approach is based on the fragment of first-order linear logic and on the bottom-up evaluation algorithm explained in Chapter 10. As discussed in Part I, our specification language has a natural correspondence with multiset rewriting systems (see in particular Section 5.4), and, in fact, it has been inspired by the approach taken in [CDL$^+$99]. In this section we introduce, rather informally, some generalities about the way we will use to specify protocols in Section 11.4.

First of all, we need a representation for the entities (e.g. principals and messages) involved. In particular, we will use a notation like

$$pr(id, s)$$

to denote a principal with identifier id and internal state s. The internal state s can store information about an ongoing execution of any given protocol (for instance, the identifier of another principal, which step of the protocol has been executed, the *role* of the principal, and so on). Typically, the state s will be a term like $init$ (indicating the initial state of a principal, before protocol execution), or a term like $step_i(data)$, where the constructor $step_i$ denotes which is the last step executed and $data$ represents the internal data of a given principal. In general, we allow more than one atom $pr(id, \_)$ inside a given configuration. In this way, we can model the possibility of a given principal to take part into different protocol runs, possibly with different *roles*.

Messages sent over a given network can in turn be represented by terms like

$$n(mess\_content)$$

where $mess\_content$ is the content of the message. Depending on the particular protocol under consideration, we can fix a specific format for messages. For instance, a message encrypted with the public key of a principal a could be represented as the term $enc(pubk(a), mess\_content)$.

Finally, we will use the Dolev-Yao intruder model (see Section 11.2.3) and the associated assumptions. In particular, as explained in Section 11.2.3, we need a way to store the intruder knowledge. We will use terms such as

$$m(inf)$$

to represent the information in possession of the intruder (m stands for the internal *memory* of the intruder). At any given instant of time, we can think of the current *state* of a given system as a *multiset* of atoms representing principals and messages currently on the network, and the intruder knowledge.

Following [CDL+99], we represent the environment in which protocol execution takes place by means of: a *protocol theory*, which includes rules for every protocol role (typically, one rule for every step of the protocol), and an *intruder theory*, which formalizes the set of possible actions of a malicious intruder who tries to break the protocol. In addition, it is possible to have additional rules for the environment. Rules assume the general format

$$\mathrm{F}_1 \otimes \ldots \otimes \mathrm{F}_n \multimap \forall \mathrm{X}_1 \ldots \forall \mathrm{X}_k.(\mathrm{G}_1 \otimes \ldots \otimes \mathrm{G}_m)$$

where F_i, G_i are atomic formulas (representing e.g. principals or messages) and X_i are variables. As explained in Section 10, the standard semantics for the universal quantifier requires new values to be chosen before application of a rule. We use this behaviour to encode nonce generation during protocol runs. As a result, we get *for free* the assumption (required by the Dolev-Yao model) that nonces are not *guessable*. In the following we will use these notational conventions: free variables inside a rule are always implicitly universally quantified, and variables are written as *upper-case* identifiers.

As far the specification of the *initial states* is concerned, we take here a slightly different approach w.r.t. the specifications in the previous chapters. Namely, we allow a *partial* specification of the initial states (see Appendix A.3). This strategy is more flexible in that it may help us to find additional hypotheses under which a given attack might take place. As a general rule, the partial specification of the initial states we have chosen requires every principal to be in his/her initial state (represented by the term *init*) at the beginning of protocol execution. As usual, we express security properties by means of their negation, i.e., using formulas representing their *violation*. We will analyze properties which can be expressed by means of *minimality conditions*.

Finally, we conclude this section by collecting together some rules which are common to *all* the examples presented in Section 11.4. In particular, we have two rules for the environment:

$$e_1) \quad pr(\mathrm{Z}, \mathrm{S}) \multimap pr(\mathrm{Z}, \mathrm{S}) \otimes \forall \mathrm{ID}.(pr(\mathrm{ID}, init))$$
$$e_2) \quad pr(\mathrm{Z}, \mathrm{S}) \multimap pr(\mathrm{Z}, \mathrm{S}) \otimes pr(\mathrm{Z}, init)$$

The first one allows creation of new principals (we use the universal quantifier to generate new identifiers for them), whereas the second rule allows creation of a new instance of a given principal (this allows a principal to start another execution of a given protocol with a new and possibly different role). Both rules can be fired at run-time, i.e., during the execution of a given protocol. We use the term *init* to denote the initial state of any given principal. We also have the following two rules for the intruder theory:

$$t_1) \quad pr(\mathrm{Z}, \mathrm{S}) \multimap pr(\mathrm{Z}, \mathrm{S}) \otimes m(\mathrm{Z})$$
$$t_2) \quad pr(\mathrm{Z}, \mathrm{S}) \multimap pr(\mathrm{Z}, \mathrm{S}) \otimes \forall \mathrm{N}.(m(\mathrm{N}))$$

The first one allows the intruder to store the identifier of any principal, whereas the second rule formalize the capability of the intruder to generate new values (e.g. nonces).

11.4 Examples

In this section we present the specification and analysis of different authentication protocols, taken from the literature on security. Some generalities concerning the encoding we will use in the following have been discussed in Section 11.3. We summarize the experimental results for this section, obtained using the tool presented in Appendix A, in Table 11.1 (see Section 11.4.5).

11.4.1 Millen's *ffgg* protocol

We begin our survey from the so-called Millen's *ffgg* protocol [Mil99]. Although an artificial protocol, its analysis will be a good starting point to illustrate the capabilities of our verification tool. In fact, this protocol provides an example of a *parallel session* attack, which requires running at least two processes for the same role. It has been proved (see [Mil99]) that no *serial* attacks exist, i.e., the protocol is secure if processes are serialized.

The protocol is as follows.

$$\begin{array}{lcl} 1.\ A \rightarrow B & : & A \\ 2.\ B \rightarrow A & : & N_1, N_2 \\ 3.\ A \rightarrow B & : & \{N_1, N_2, S\}_{K_b} \% \{N_1, X, Y\}_{K_b} \\ 4.\ B \rightarrow A & : & N_1, X, \{X, Y, N_1\}_{K_b} \end{array}$$

As usual, N_1 and N_2 stand for nonces, created by principal B and included in message 2. The $m\%m'$ notation, introduced in [Low98], used in message 3 represents a message which has been created by the sender according to format m, but is interpreted as m' by the receiver. In this case, the intuition is that upon receiving message 3, B checks that the first component does correspond to the first of the two nonces previously created, while no check at all is performed on the second component of the message. In message 2, S stands for a secret, of the same length as a nonce, which is in possession of B. The security property one is interested to analyze is whether the secret S can be disclosed to a malicious intruder.

We have implemented the *ffgg* protocol through the specification shown in Figure 11.1, while the intruder theory is presented in Figure 11.2. The specification consists of a set of protocol rules (rules p_1 through p_4 in Figure 11.1) and an intruder theory (rules i_1 through

$p_1)$ $pr(\mathrm{A}, init) \parr pr(\mathrm{B}, init) \multimap pr(\mathrm{A}, step1(\mathrm{B})) \parr pr(\mathrm{B}, init) \parr n(plain(\mathrm{A}))$

$p_2)$ $pr(\mathrm{B}, init) \parr n(plain(\mathrm{A})) \multimap \forall \mathrm{N1}.\forall \mathrm{N2}.(pr(\mathrm{B}, step2(\mathrm{A}, \mathrm{N1})) \parr n(plain(\mathrm{N1}, \mathrm{N2})))$

$p_3)$ $pr(\mathrm{A}, step1(\mathrm{B})) \parr n(plain(\mathrm{N1}, \mathrm{N2})) \multimap \forall \mathrm{S}.(pr(\mathrm{A}, step3(\mathrm{B}, \mathrm{S})) \parr n(enc(pubk(\mathrm{B}), \mathrm{N1}, \mathrm{N2}, \mathrm{S})))$

$p_4)$ $pr(\mathrm{B}, step2(\mathrm{A}, \mathrm{N1})) \parr n(enc(pubk(\mathrm{B}), \mathrm{N1}, \mathrm{X}, \mathrm{Y})) \multimap pr(\mathrm{B}, step4(\mathrm{A})) \parr n(plain(\mathrm{N1}, \mathrm{X}), enc(pubk(\mathrm{B}), \mathrm{X}, \mathrm{Y}, \mathrm{N1}))$

Figure 11.1: Specification of the *ffgg* protocol

i_8 in Figure 11.2). We remind the reader that the four rules e_1, e_2, t_1 and t_2 discussed in Section 11.3 are *in addition* to the present rules.

Protocol rules directly correspond to the informal description of the *ffgg* protocol previously presented. We have followed the conventions outlined in Section 11.3 to model the internal state of principals. In particular, we have a term *init* denoting the initial state of a principal, and the constructors *step*1, *step*2, *step*3 and *step*4 to model the different steps of a protocol run. At every step, each principal needs to remember the identifier of the other principal he/she is executing the protocol with. In addition, at step 2 the responder stores the first nonce created (in order to be able to perform the required check, see rule p_4), and at step 3 the initiator of the protocol remembers the secret S. We have modeled the secret S using the universal quantifier, as for nonces. In this way, we can get for free the requirement that the secret initially is only known to the principal who possesses it. Finally, we have term constructors *plain*(...) and *enc*(...) (to be precise, we should say a *family* of term constructors, we find it convenient to overload the same symbol with different *arities*) to distinguish *plain* messages from *encrypted* messages.

The intruder theory is made up of rules i_1 through i_8 in Figure 11.2. It is an instance of the Dolev-Yao intruder theory illustrated in Section 11.2.3. Let us discuss it in more detail. Rules i_1 through i_4 are *decomposition* rules, whereas rules i_5 through i_8 are *composition* rules. We have four rules for each of the two different kinds (composition and decomposition) of messages, dealing with the different formats of messages used in the *ffgg* protocol. For instance, rule i_1 deals with decomposition of plain messages with one component, whereas rule i_4 deals with decomposition of messages with two plain components and one encrypted component, and so on. Clearly, the intruder cannot furtherly decompose encrypted components, which are stored exactly as they are, whereas plain messages are decomposed into their atomic constituents. The intruder theory we have presented is an instance of the general Dolev-Yao intruder theory, in that intruder rules have been tailored to the particular form of messages used in the specific protocol under consideration. In general, we have followed this strategy also for the other examples we will present

$i_1)$ $n(plain(\mathrm{X})) \multimap m(plain(\mathrm{X}))$

$i_2)$ $n(plain(\mathrm{X},\mathrm{Y})) \multimap m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y}))$

$i_3)$ $n(enc(\mathrm{X},\mathrm{Y},\mathrm{Z},\mathrm{W})) \multimap m(enc(\mathrm{X},\mathrm{Y},\mathrm{Z},\mathrm{W}))$

$i_4)$ $n(plain(\mathrm{X},\mathrm{Y}), enc(\mathrm{U},\mathrm{V},\mathrm{W},\mathrm{Z})) \multimap m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \parr m(enc(\mathrm{U},\mathrm{V},\mathrm{W},\mathrm{Z}))$

$i_5)$ $m(plain(\mathrm{X})) \multimap m(plain(\mathrm{X})) \parr n(plain(\mathrm{X}))$

$i_6)$ $m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \multimap m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \parr n(plain(\mathrm{X},\mathrm{Y}))$

$i_7)$ $m(enc(\mathrm{X},\mathrm{Y},\mathrm{Z},\mathrm{W})) \multimap m(enc(\mathrm{X},\mathrm{Y},\mathrm{Z},\mathrm{W})) \parr n(enc(\mathrm{X},\mathrm{Y},\mathrm{Z},\mathrm{W}))$

$i_8)$ $m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \parr m(enc(\mathrm{U},\mathrm{V},\mathrm{W},\mathrm{Z})) \multimap m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \parr m(enc(\mathrm{U},\mathrm{V},\mathrm{W},\mathrm{Z})) \parr n(plain(\mathrm{X},\mathrm{Y}), enc(\mathrm{U},\mathrm{V},\mathrm{W},\mathrm{Z}))$

Figure 11.2: Intruder theory for the *ffgg* protocol

in the following. We refer to Section 11.5 for a discussion on how this approach could be generalized. In the case of *ffgg* protocol, we have not given the intruder any capability to decrypt messages. This hypothesis can be relaxed (see for instance the analysis of Needham-Schroeder protocol in Section 11.4.2). The present specification is sufficient for our purposes.

Now, we can simply specify the set of unsafe states as follows:

$$u)\ \ pr(alice, step3(bob, \mathrm{S})) \parr pr(bob, step4(alice)) \parr m(plain(\mathrm{S})) \multimap \top$$

Namely, we consider a configuration unsafe if there exist two honest principals, say *alice* and *bob*, who have run the protocol to completion (i.e., they have completed, respectively, step 4 and step 3) and the secret S has been disclosed to the intruder (i.e., it is eventually stored in the intruder's internal memory).

Running our verification algorithm, we automatically find an attack, whose trace is shown in Figure 11.3. Besides the usual notational conventions, we have fixed the following shortened notations: *alice* has been shortened to *al*; p_d^i stands for a principal p after execution of step number i with internal data d; p^{init} stands for a principal in its initial state; we have omitted the *plain* term constructor for plain messages; we have noted encrypted messages using the usual protocol notation; $\mathcal{M}(x, y, \ldots)$ stands for the multiset $m(x), m(y), \ldots$; finally, $\Sigma_1 = \Sigma, n1, n2$, $\Sigma_2 = \Sigma, n1, n2, n3, n4$, and $\Sigma_3 = \Sigma, n1, n2, n3, n4, s$. The attack is exactly the *parallel session* one described in [Mil99]. We note that this attack is also an example of a *type flaw* attack, in that it relies on the secret S be passed as a nonce (under the hypothesis that the lengths of the respective fields are the same).

We also remark that we have performed some further experiments, which we don't discuss

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash_{\Sigma_3} \top, bob^4_{al}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}, \{s, n1, n3\}_{K_{bob}})}\top_r}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, m(al, n1, n2, n3, n4, s, \{n3, s, n1\}_{K_{bob}}, \{s, n1, n3\}_{K_{bob}})}bc^{(u)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^4_{al}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}), n(n3, s, \{s, n1, n3\}_{K_{bob}})}bc^{(i_4)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^4_{al}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}), n(\{n3, s, n1\}_{K_{bob}})}bc^{(p_4)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}), n(\{n3, s, n1\}_{K_{bob}})}bc^{(i_7)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}})}bc^{(i_4)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(n1, n3, \{n3, s, n1\}_{K_{bob}})}bc^{(p_4)}}{P \vdash_{\Sigma_3} al^3_{bob,s}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(\{n1, n3, s\}_{K_{bob}})}bc^{(p_3)}}{P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(n1, n3)}bc^{(i_6)}}{P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4)}bc^{(i_2)}}{P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2), n(n3, n4)}bc^{(p_2)}}{P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al, n1, n2), n(al)}bc^{(i_5)}}{P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al, n1, n2)}bc^{(i_2)}}{P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al), n(n1, n2)}bc^{(p_2)}}{P \vdash_{\Sigma} al^1_{bob}, bob^{init}, bob^{init}, m(al), n(al)}bc^{(p_1)}}{P \vdash_{\Sigma} al^{init}, bob^{init}, bob^{init}, m(al)}bc^{(t_1)}}{P \vdash_{\Sigma} al^{init}, bob^{init}, bob^{init}}bc^{(e_2)}}{P \vdash_{\Sigma} al^{init}, bob^{init}}$$

Figure 11.3: A parallel session attack to the *ffgg* protocol

in detail for the sake of brevity. In particular, we wanted to ascertain the role of the two nonces N_1 and N_2 in the *ffgg* protocol. According to the informal notation for the protocol introduced at the beginning of this section, principal B only checks that the first component of message T is the nonce N_1, whereas no check is performed for the second component. We have verified that imposing the check on the second component, the *ffgg* protocol is safe w.r.t. the security property and the intruder theory we have presented, while removing all checks introduces *serial* attacks.

We think that this example is a good illustration of the capabilities of our verification tool. In fact, using the *backward* evaluation strategy championed in this thesis, we are able to automatically find a parallel session attack, *without encoding any prior knowledge* about the kind of attacks to look for. In particular, according to [Mil99] the *ffgg* can be generalized to protocols which only admit *higher-order* parallel attacks (i.e., attacks which take place only in presence of three or more concurrent roles for the same principal). Using our tool we can automatically find such attacks, if any exists. This distinguishes our methodology from most approaches based on model-checking, which operate on a finite-state abstraction of a given protocol, and require the number of principals and the number of roles to be fixed *in advance*.

11.4.2 The Needham-Schroeder Protocol

In this section we analyze the Needham-Schroeder public-key authentication protocol, previously discussed in Section 11.2.4. For the sake of precision, we restrict our attention the fragment of the Needham-Schroeder protocol where the key distribution phase (i.e., the messages exchanged with the trusted server) has been omitted.

The resulting protocol, corresponding to messages 3, 6, and 7 of Section 11.2.4, is as follows:

$$\begin{array}{lcl} 1. \quad A \rightarrow B & : & \{N_a, A\}_{K_b} \\ 2. \quad B \rightarrow A & : & \{N_a, N_b\}_{K_a} \\ 3. \quad A \rightarrow B & : & \{N_b\}_{K_b} \end{array}$$

and has been implemented using the specification illustrated in Figure 11.4 and the intruder theory in Figure 11.5. Let us discuss the rules in more detail.

The protocol rules have been encoded in the same way as for the *ffgg* protocol of Section 11.4.1, and directly correspond to the informal notation previously introduced. This time, all messages sent over the network are encrypted, therefore we have modeled them using atomic formulas like $n(pubk(id), mess\_content)$, where id is a principal identifier, whereas $mess\_content$ is a term of the form $mess(\ldots)$. Internal states of principals have been enriched in order to express the security violations we will present in the following.

The intruder theory is shown in Figure 11.5. We consider here a more extended intruder

$p_1)$ $pr(\mathrm{A}, init) \parr pr(\mathrm{B}, init) \circ\!\!- pr(\mathrm{B}, init) \parr \forall \mathrm{NA}.(pr(\mathrm{A}, step1(\mathrm{NA}, \mathrm{B})) \parr n(pubk(\mathrm{B}), mess(\mathrm{NA}, \mathrm{A})))$

$p_2)$ $pr(\mathrm{B}, init) \parr n(pubk(\mathrm{B}), mess(\mathrm{NA}, \mathrm{A})) \circ\!\!- \forall \mathrm{NB}.(pr(\mathrm{B}, step2(\mathrm{NA}, \mathrm{NB}, \mathrm{A})) \parr n(pubk(\mathrm{A}), mess(\mathrm{NA}, \mathrm{NB})))$

$p_3)$ $pr(\mathrm{A}, step1(\mathrm{NA}, \mathrm{B})) \parr n(pubk(\mathrm{A}), mess(\mathrm{NA}, \mathrm{NB})) \circ\!\!- pr(A, step3(\mathrm{NA}, \mathrm{NB}, \mathrm{B})) \parr n(pubk(\mathrm{B}), mess(\mathrm{NB}))$

$p_4)$ $pr(\mathrm{B}, step2(\mathrm{NA}, \mathrm{NB}, \mathrm{A})) \parr n(pubk(\mathrm{B}), mess(\mathrm{NB})) \circ\!\!- pr(\mathrm{B}, step4(\mathrm{NA}, \mathrm{NB}, \mathrm{A}))$

Figure 11.4: Specification of the Needham-Schroeder protocol

theory w.r.t. the one for the *ffgg* protocol in Figure 11.2. Namely, we give the intruder the capability to decrypt messages addressed to himself/herself. Given that private keys are not exchanged, the intruder will never be able to know the private keys of some other principal, therefore we still assume that the intruder is not able to decrypt messages other than the ones intended for himself/herself. Rule i_1 and i_2 deal with *interception*: the intruder can intercept any message and replay it (possibly more than once) at any given time in the future. Rule i_3 states that the intruder can *decompose* any message addressed to himself/herself. Rules i_4 and i_5 are the usual rules for *decomposition* of messages (with one or two components). Rules i_6 and i_7 are the corresponding rules for *composition* of messages (with one or two components). Finally, using rule i_8 the intruder can send a newly composed message to *any* principal.

Now, the specification of unsafe states is as follows:

$$
\begin{array}{ll}
u_1) & pr(alice, step3(\mathrm{NA}, \mathrm{NB}, bob)) \parr m(\mathrm{NA}) \circ\!\!- \top \\
u_2) & pr(alice, step3(\mathrm{NA}, \mathrm{NB}, bob)) \parr m(\mathrm{NB}) \circ\!\!- \top \\
u_3) & pr(bob, step4(\mathrm{NA}, \mathrm{NB}, alice)) \parr m(\mathrm{NA}) \circ\!\!- \top \\
u_4) & pr(bob, step4(\mathrm{NA}, \mathrm{NB}, alice)) \parr m(\mathrm{NB}) \circ\!\!- \top
\end{array}
$$

Namely, a state is unsafe if there exist two principals, say *alice* and *bob*, such that either *alice* has run the protocol to completion with *bob*, or *bob* with *alice*, and *at least* one of the two nonces has been disclosed to the intruder. We call the security property specified by the above rules *strong correctness*.

We can now run our verification tool on the resulting specification. As observed by Lowe [Low95], Needham-Schroeder protocol is not safe w.r.t. the above security properties. In fact, we find the attack shown in Figure 11.6, corresponding to the one presented in [Low95] (we have followed the same conventions as in Figure 11.3, with $\Sigma_1 = \Sigma, na$ and $\Sigma_2 = \Sigma, na, nb$). The attack takes place because *alice* decides to contact the intruder, *without knowing he/she is cheating*. Thus, the intruder is able to impersonate *alice* and

$i_1)$ $n(pubk(\mathrm{A}), \mathrm{M}) \multimap intercept(n(pubk(\mathrm{A}), \mathrm{M}))$

$i_2)$ $intercept(n(pubk(\mathrm{A}), \mathrm{M})) \multimap intercept(n(pubk(\mathrm{A}), \mathrm{M})) \parr n(pubk(\mathrm{A}), \mathrm{M})$

$i_3)$ $n(pubk(intruder), \mathrm{M}) \multimap dec(\mathrm{M})$

$i_4)$ $dec(mess(\mathrm{M})) \multimap m(\mathrm{M})$

$i_5)$ $dec(mess(\mathrm{M}, \mathrm{N})) \multimap m(\mathrm{M}) \parr m(\mathrm{N})$

$i_6)$ $m(\mathrm{M}) \multimap m(\mathrm{M}) \parr comp(mess(\mathrm{M}))$

$i_7)$ $m(\mathrm{M}) \parr m(\mathrm{N}) \multimap m(\mathrm{M}) \parr m(\mathrm{N}) \parr comp(mess(\mathrm{M}, \mathrm{N}))$

$i_8)$ $comp(\mathrm{C}) \multimap n(pubk(\mathrm{Z}), \mathrm{C})$

Figure 11.5: Intruder theory for the Needham-Schroeder protocol

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{P \vdash_{\Sigma_2} \top, al^3_{na,nb,i}, i^{init}, m(al, nb)}\,\top_r
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^4_{na,nb,al}, i^{init}, m(na, al, nb)}\,bc^{(u_3)}
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^2_{na,nb,al}, i^{init}, m(na, al, nb), n(\{mess(nb)\}_{K_{bob}})}\,bc^{(p_4)}
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^2_{na,nb,al}, i^{init}, m(na, al, nb), comp(mess(nb))}\,bc^{(i_8)}
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^2_{na,nb,al}, i^{init}, m(na, al, nb)}\,bc^{(i_6)}
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^2_{na,nb,al}, i^{init}, m(na, al), dec(mess(nb))}\,bc^{(i_4)}
}{P \vdash_{\Sigma_2} al^3_{na,nb,i}, bob^2_{na,nb,al}, i^{init}, m(na, al), n(\{mess(nb)\}_{K_i})}\,bc^{(i_3)}
}{P \vdash_{\Sigma_2} al^1_{na,i}, bob^2_{na,nb,al}, i^{init}, m(na, al), n(\{mess(na, nb)\}_{K_{al}})}\,bc^{(p_3)}
}{P \vdash_{\Sigma_1} al^1_{na,i}, bob^{init}, i^{init}, m(na, al), n(\{mess(na, al)\}_{K_{bob}})}\,bc^{(p_2)}
}{P \vdash_{\Sigma_1} al^1_{na,i}, bob^{init}, i^{init}, m(na, al), comp(mess(na, al))}\,bc^{(i_8)}
}{P \vdash_{\Sigma_1} al^1_{na,i}, bob^{init}, i^{init}, m(na, al)}\,bc^{(i_7)}
}{P \vdash_{\Sigma_1} al^1_{na,i}, bob^{init}, i^{init}, dec(mess(na, al))}\,bc^{(i_5)}
}{P \vdash_{\Sigma_1} al^1_{na,i}, bob^{init}, i^{init}, n(\{mess(na, al)\}_{K_i})}\,bc^{(i_3)}
}{P \vdash_{\Sigma} al^{init}, bob^{init}, i^{init}}\,bc^{(p_1)}
$$

Figure 11.6: An attack to the Needham-Schroeder protocol

cheat *bob*. Note that as a result the protocol has been broken from the point of view of *bob*. In fact, *bob* thinks he has got authentication with *alice* and that provided *alice* is honest, the nonces have not been disclosed to anyone else (which is false), whereas from the point of view of *alice*, she correctly thinks to have established authentication with the intruder (the nonces have been disclosed to *bob*, but only because the intruder is cheating and *alice* does not know that).

With this in mind, we can now try the following *stronger* security violations (we call the corresponding security property *weak correctness*).

$$u'_1) \quad pr(alice, step3(\text{NA}, \text{NB}, bob)) \parr pr(bob, step4(\text{NA}, \text{NB}, alice)) \parr m(\text{NA}) \multimap \top$$
$$u'_2) \quad pr(alice, step3(\text{NA}, \text{NB}, bob)) \parr pr(bob, step4(\text{NA}, \text{NB}, alice)) \parr m(\text{NB}) \multimap \top$$

Namely, we try to ascertain whether it is possible that two honest principals *alice* and *bob* *both* believe to have completed the protocol with each other, and still *at least* one of the two nonces has been disclosed to the intruder (as the reader can easily verify, this is not the case for the trace of the previous attack). This time the verification algorithm terminates, proving that Needham-Schroeder protocol is safe with respect to this property.

We conclude this section by showing how the methodology of *invariant strengthening*, discussed in Section 6.1.2.1, can improve the verification algorithm performance. Namely, we augment the set of security violations with this further rule:

$$u'_3) \quad pr(alice, step1(\text{NA}, bob)) \parr m(\text{NA}) \multimap \top$$

Intuitively, this violation is never met. In fact, the nonce NA is created by *alice* during step 1 and sent to *bob*. If *bob* is honest, he will never send it to the intruder. Adding this rule has the effect of accelerating convergence of the fixpoint computation (see the experimental results in Table 11.1). Note that the following invariant is instead violated (the attack follows the same scheme of the one in Figure 11.6):

$$u'_4) \quad pr(bob, step2(\text{NA}, \text{NB}, alice)) \parr m(\text{NB}) \multimap \top$$

We remind that adding rules (including axioms) to the theory is always sound, in the sense that if no attack is found in the augmented theory, no attack can be found in the original one.

11.4.3 Corrected Needham-Schroeder

As observed by Lowe [Low95], the Needham-Schroeder protocol can be fixed with a small modification. The problem with the original protocol is that the second message exchanged does not contain the identity of the responder. Adding the responder's identity to this

message prevents the intruder from replaying it, because now the initiator is expecting a message from the intruder. The corrected version of Needham-Schroeder protocol is

$$\begin{array}{lcl} 1. & A \rightarrow B & : \{N_a, A\}_{K_b} \\ 2. & B \rightarrow A & : \{B, N_a, N_b\}_{K_a} \\ 3. & A \rightarrow B & : \{N_b\}_{K_b} \end{array}$$

We need to make minor modifications to our previous specification. Namely, we modify rules p_2) and p_3) by adding the additional argument B:

$$\begin{array}{ll} p'_2) & pr(\mathrm{B}, init) \parr n(pubk(\mathrm{B}), mess(\mathrm{NA}, \mathrm{A})) \circ\!\!- \forall \mathrm{NB}.(pr(\mathrm{B}, step2(\mathrm{NA}, \mathrm{NB}, \mathrm{A})) \parr \\ & \quad n(pubk(\mathrm{A}), mess(\mathrm{B}, \mathrm{NA}, \mathrm{NB}))) \\ p'_3) & pr(\mathrm{A}, step1(\mathrm{NA}, \mathrm{B})) \parr n(pubk(\mathrm{A}), mess(\mathrm{B}, \mathrm{NA}, \mathrm{NB})) \circ\!\!- \\ & \quad pr(A, step3(\mathrm{NA}, \mathrm{NB}, \mathrm{B})) \parr n(pubk(\mathrm{B}), mess(\mathrm{NB})) \end{array}$$

and we add two rules for composition and decomposition of messages with three components:

$$\begin{array}{ll} u_9) & dec(mess(\mathrm{M}, \mathrm{N}, \mathrm{O})) \circ\!\!- m(\mathrm{M}) \parr m(\mathrm{N}) \parr m(\mathrm{O}) \\ u_{10}) & m(\mathrm{M}) \parr m(\mathrm{N}) \parr m(\mathrm{O}) \circ\!\!- m(\mathrm{M}) \parr m(\mathrm{N}) \parr m(\mathrm{O}) \parr comp(mess(\mathrm{M}, \mathrm{N}, \mathrm{O})) \end{array}$$

We can now use our algorithm to automatically verify if the protocol satisfy *strong correctness* (axioms u_1 through u_4). As in Section 11.4.2, we can accelerate convergence by means of invariant strengthening (see the experimental results in Table 11.1). We can use the two invariants u'_3 and u'_4 discussed in Section 11.4.2, which should now *both* hold. The verification algorithm terminates, as expected, proving that the modified version of the Needham-Schroeder protocol is correct w.r.t. the notion of *strong* correctness.

11.4.4 The Otway-Rees Protocol

The Otway-Rees protocol [OR87] provides a typical example of a *type flaw attack*. It is intended for *key distribution* between two principals communicating with a central server by means of shared keys (the protocol assumes *symmetric key encryption*, see Section 11.2.1).

The protocol is as follows (the form presented here is the one given in [BAN89]).

$$\begin{array}{lcl} 1. & A \rightarrow B & : N, A, B, \{N_a, N, A, B\}_{K_{as}} \\ 2. & B \rightarrow S & : N, A, B, \{N_a, N, A, B\}_{K_{as}}, \{N_b, N, A, B\}_{K_{bs}} \\ 3. & S \rightarrow B & : N, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}} \\ 4. & B \rightarrow A & : N, \{N_a, K_{ab}\}_{K_{as}} \end{array}$$

The protocol is run between two principals A and B, communicating with a *trusted server* S by means of shared keys K_{as} and K_{bs}. The purpose of the protocol is to get a new key K_{ab},

p_1) $pr(\mathrm{A}, init) \parr pr(\mathrm{B}, init) \multimap pr(\mathrm{B}, init) \parr \forall \mathrm{N}.\forall \mathrm{NA}.$
$(n(plain(cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B})))) \parr$
$pr(\mathrm{A}, step1(\mathrm{B}, \mathrm{N}, \mathrm{NA})))$

p_2) $n(plain(cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))))) \parr$
$pr(\mathrm{B}, init) \multimap \forall \mathrm{NB}.(pr(\mathrm{B}, step2(\mathrm{A}, \mathrm{N}, \mathrm{NB})) \parr$
$n(plain(cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B})))),$
$enc(sk(\mathrm{B}, s), cons(\mathrm{NB}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B})))))$

p_3) $n(plain(cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B})))),$
$enc(sk(\mathrm{B}, s), cons(\mathrm{NB}, cons(\mathrm{N}, cons(\mathrm{A}, \mathrm{B}))))) \parr pr(s, init) \multimap pr(s, init) \parr$
$\forall \mathrm{KAB}.n(plain(\mathrm{N}), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, \mathrm{KAB})), enc(sk(\mathrm{B}, s), cons(\mathrm{NB}, \mathrm{KAB})))$

p_4) $n(plain(\mathrm{N}), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, \mathrm{KAB})), enc(sk(\mathrm{B}, s), cons(\mathrm{NB}, \mathrm{KAB}))) \parr$
$pr(\mathrm{B}, step2(\mathrm{A}, \mathrm{N}, \mathrm{NB})) \multimap pr(\mathrm{B}, step3(\mathrm{A}, \mathrm{KAB})) \parr$
$n(plain(\mathrm{N}), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, \mathrm{KAB})))$

p_5) $pr(\mathrm{A}, step1(\mathrm{B}, \mathrm{N}, \mathrm{NA})) \parr n(plain(\mathrm{N}), enc(sk(\mathrm{A}, s), cons(\mathrm{NA}, \mathrm{KAB}))) \multimap$
$pr(\mathrm{A}, step4(\mathrm{B}, \mathrm{KAB}))$

Figure 11.7: Specification of the Otway-Rees protocol

generated by the trusted server, to be used as a shared key in subsequent communications between A and B. At the first step, principal A generates a nonce N, to be used as a *run identifier*, and a nonce N_a, and sends to B the plaintext N, A, B and an encrypted message, readable only by the server S, of the form shown. In turn, principal B generates a nonce N_b and forwards A's message to S, together with a similar encrypted component. The server checks that the N, A, B components in both messages match, and, if so, generates a new key K_{ab} and replies to B with message 3 above, which includes a component intended for B and one for A. The component intended for A is forwarded to him/her by B with message 4.

We have encoded the Otway-Rees protocol with the rules in Figure 11.7. Let us discuss them in more detail. The encoding is similar to the one used for Millen's *ffgg* protocol (see Section 11.4.1), in particular we distinguish between the plaintext part of messages (term constructor *plain*) and the encrypted part (term constructor *enc*). Furthermore, we use a concatenation operator *cons* to glue together different components inside the plaintext or the ciphertext. In this way, we are able to capture type flaw attacks. We use an identifier s for the trusted server, and the notation $sk(id, s)$ to denote the shared key between principal id and s. Rules p_1 through p_5 are a direct translation of the protocol steps written in the usual notation. As usual, we have denoted the internal state of principals by means of term constructors like $step_i$. When the protocol is run to completion, the internal state of

$$
\begin{array}{ll}
i_1) & n(plain(\mathrm{X}), enc(sk(\mathrm{K}, s), \mathrm{Y})) \multimap m(plain(\mathrm{X})) \parr m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \\
i_2) & m(plain(\mathrm{X}), enc(sk(\mathrm{K}, s), \mathrm{Y}), enc(sk(\mathrm{H}, s), \mathrm{W})) \multimap m(plain(\mathrm{X})) \parr \\
& \quad m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \parr m(enc(sk(\mathrm{H}, s), \mathrm{W})) \\
i_3) & m(plain(cons(\mathrm{X}, \mathrm{Y}))) \multimap m(plain(cons(\mathrm{X}, \mathrm{Y}))) \parr m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \\
i_4) & m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \multimap m(plain(\mathrm{X})) \parr m(plain(\mathrm{Y})) \parr \\
& \quad m(plain(cons(\mathrm{X}, \mathrm{Y}))) \\
i_5) & m(plain(\mathrm{X})) \parr m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \multimap m(plain(\mathrm{X})) \parr m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \parr \\
& \quad n(plain(\mathrm{X}), enc(sk(\mathrm{K}, s), \mathrm{Y})) \\
i_6) & m(plain(\mathrm{X})) \parr m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \parr m(enc(sk(\mathrm{H}, s), \mathrm{W})) \multimap m(plain(\mathrm{X})) \parr \\
& \quad m(enc(sk(\mathrm{K}, s), \mathrm{Y})) \parr m(enc(sk(\mathrm{H}, s), \mathrm{W})) \parr \\
& \quad n(plain(\mathrm{X}), enc(sk(\mathrm{K}, s), \mathrm{Y}), enc(sk(\mathrm{H}, s), \mathrm{W}))
\end{array}
$$

Figure 11.8: Intruder theory for the Otway-Rees protocol

each of the two involved principals contains the identity of the other principal and the new shared key obtained from the server.

The intruder theory in presented in Figure 11.8. Rules i_1 and i_2 are the usual *decomposition* rules for, respectively, messages with one encrypted component and with two encrypted components. Rules i_3 and i_4 allow the intruder to arbitrarily decompose and re-assemble plaintext messages, whereas encrypted messages are stored as they are. Finally, rules i_5 and i_6 allow the intruder to create new messages from stored components.

We wish to verify if the intruder can get the shared key which comes from a protocol run between two honest principals, say *alice* and *bob*. The specification of unsafe states is straightforward:

$$
\begin{array}{ll}
u_1) & pr(bob, step3(alice, \mathrm{KAB})) \parr m(plain(\mathrm{KAB})) \multimap \top \\
u_2) & pr(alice, step4(bob, \mathrm{KAB})) \parr m(plain(\mathrm{KAB})) \multimap \top
\end{array}
$$

Running our verification tool, we automatically find the type flaw attack described in [CJ97]. The corresponding trace is shown in Figure 11.9 (we have followed the usual conventions, with $\Sigma_1 = \Sigma, n1, na$). The attack takes place because a malicious intruder can intercept the first message, and, after stripping it of the A and B components (in the plaintext part), replay it as the last message of the protocol. The attack is successful under the hypothesis that the triple N, A, B may be erroneously accepted, by the initiator of the protocol, as the desired key. This is clearly a security flaw because the triple N, A, B is sent in clear in the first message, and therefore publicly known.

The Otway-Rees protocol provides a classical example of a type flaw attack. This kind

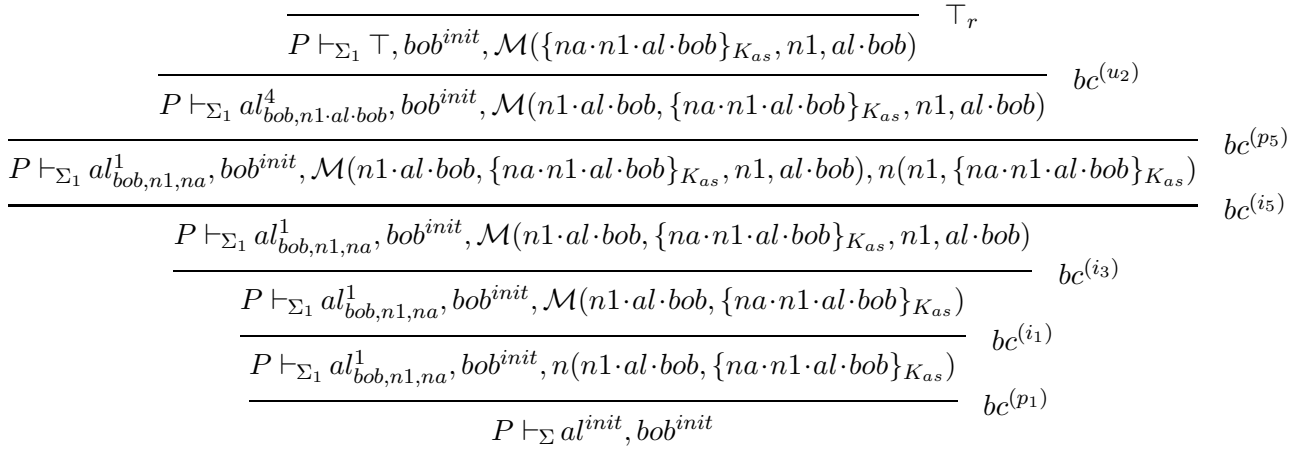

Figure 11.9: An attack to the Otway-Rees protocol

Protocol	**Invar**	**Steps**	**Size**	**MSize**	**Time**	**Verified**
Millen's ffgg		14	306	677	1335	*attack*
Millen's ffgg (*corrected*)		14	27	810	2419	*yes*
Needham-Schroeder (*strong correctness*)		13	294	323	45	*attack*
Needham-Schroeder (*weak correctness*)		13	304	755	516	*yes*
	$\surd$	12	299	299	63	*yes*
Corrected Needham-Schroeder (*strong correctness*)		14	1575	1575	791	*yes*
	$\surd$	9	402	402	31	*yes*
Otway-Rees		5	10339	10339	4272	*attack*

Table 11.1: Analysis of authentication protocols: experimental results

of attacks are very pervasive in authentication. Another classical example of type-flawed protocol is the Yahalom protocol [CJ97].

11.4.5 Experimental Results

In Table 11.1 we summarize the experimental results for the examples of Section 11.4. We use the following conventions: a symbol $\surd$ in column 'Invar' indicates use of the invariant strengthening technique; the column 'Steps' denotes the number of iterations needed to reach a fixpoint (or before finding an attack); 'Size' is the number of multisets contained in the fixpoint (or at the time the attack was found), while 'MSize' is the maximum number

of multisets computed at any step; 'Time' is the execution time (in seconds).

11.5 Future Work

In this section we discuss some lines of research for future work. We refer to Appendix A for a discussion on the verification tool we have used to carry out the experiments discussed in this chapter, and on the future improvements we plan to implement. As an example, in order to perform protocol correctness analysis, we would like to support automatic or semi-automatic generation of invariants and pruning conditions. On the theoretical side, we have also singled out the following points.

We want to study a (possibly automatic) translation between the usual informal description of protocols and our representation. As suggested by the examples presented in this chapter, a one-to-one translation (one rule for every step) could be sufficient, provided we have a way to store the information about the internal state of principals.

Furthermore, we want to study a general formulation of the Dolev-Yao intruder theory, which could be optimized and *selectively* refined for analyzing the specific protocol at hand. This refinement and optimization phase can be useful in order to analyze a given protocol under different assumptions for the intruder capabilities, and it is also crucial for efficiency considerations, as exemplified by the experiments we have presented. Specifically, optimizing the rules for composition and decomposition of messages seems to be crucial to improve the verification algorithm performance. We plan to use techniques like *folding/unfolding* to automatize this process. A similar technique is used for instance in [Bla01].

We also consider the possibility of enriching the multiset rewriting formalism underlying our specification language with a more general *term* rewriting theory based on *equational theories* and AC unification, as done for instance in [Mea96, DMT98, JRV00].

We want to carry out further experiments on other security protocols presented in the literature. In particular, we plan to combine our verification tool with a constraint solver, along the lines discussed in Chapter 9, to analyze protocols which make use of *timestamps* to ensure freshness of messages exchanged between principals. Other classes of protocols which could be worth investigating include *fair exchange* protocols and *looping* protocols like Kerberos [BAN89].

Another topic we would like to investigate is *typed multiset rewriting* [Cer01b], which extends multiset rewriting with a typing theory based on dependent types with subsorting. Dependent types can be used to enforce dependency between an encryption key and its owner. The paper [Cer01b] also presents some extensions which increase the flexibility of multiset rewriting specifications, e.g. using memory predicates to remember information across role executions.

Finally, an open question is the problem of non-termination. In the few examples we have presented, our algorithm is always terminating, even without invariant strengthening. Although secrecy has been proved to be undecidable, even for finite-length protocols with data of bounded complexity [CDL+99], one may ask if a more restricted subclass of protocols exists, for which the verification algorithm presented here is always terminating.

11.6 Related Work

In this chapter we have presented security protocols as a possible application field for our methodology based on a multiset-rewriting-like specification language and on a backward search verification strategy. Our verification procedure is tailored to study security violations which can be specified by means of *minimality conditions*. While this may rule out interesting properties, e.g. questions of *belief* [BAN89], the proposed approach can be used to study secrecy and confidentiality properties. No artificial limit is imposed on the number of simultaneous sessions we are able to analyze.

We have performed some experiments on different authentication protocols which show that the methodology we propose can be effective either to find *attacks* or to validate existing protocols. We plan to overcome some current limitations of our approach, in particular we plan to refine and automatize the specification phase of protocols and of the intruder theory, as discussed in Section 11.5. We expect further results from future work on this point. Below we discuss some related work in more detail.

A wide research area in security protocol analysis is related to rewriting logic [Mes92]. For instance, we mention [Cir01], which specifies security protocols as rewriting theories which can be executed in the ELAN system [BKK+96]. A similar approach is followed in [DMT98], where the target executable language is instead Maude [CELM96]. Perhaps the more interesting work in this class is [JRV00]. This work presents an automatic compilation process from security protocol descriptions into rewrite rules. The resulting specifications are then executed using the daTac theorem prover [Vig95]. As a difference with [DMT98], which is based on *matching*, the execution strategy of [JRV00] relies on *narrowing* and AC unification. Our approach, based on multiset unification, is clearly closer to the latter approach, although currently we do not support equational theories. All of the above approaches are limited to protocol debugging, therefore they can find attacks mounted on a given protocol, but they cannot be used to analyze correctness. Also, a crucial difference is that all the above works are based on a *forward* breadth-first-search strategy, while effectiveness of our verification algorithm strongly relies on a *backward* search strategy. Another approach which shares some similarity with ours is [Del01], where a specification for security protocols based on rewriting and encoded in a subset of intuitionistic logic is presented. The author uses universal quantification to generate nonces, like us, and

embedded implication to store the knowledge of agents. This approach is still limited to protocol debugging.

An alternative approach to verifying security protocols is based on model checking. For instance, the FDR model checking tool was used by Lowe [Low96] to analyze the Needham-Schroeder public-key protocol and the attack previously reported in [Low95]. Other works which fall into this class are [MCJ97, RB99]. All these approaches have in common the use of some kind of abstraction to transform the original problem into a *finite-state* model-checking problem, which is then studied by performing a *forward* reachability analysis. Using a finite-state approximation has the advantage of guaranteeing termination, however it only allows one to analyze a fixed number of concurrent protocol runs, an approach which is infeasible as this number increases. As a difference, we use a *symbolic* representation for *infinite* sets of states and a *backward* reachability verification procedure, which avoid putting limitations on the number of parallel sessions we are able to analyze.

Theorem proving techniques are used in [Pau98], where protocols are inductively defined as sets of traces, and formally analyzed using the theorem prover Isabelle [Pau94]. Here, analysis is a *semi-automatic* process which can take several days. The NRL protocol analyzer [Mea96] provides a mixed approach. It is based on protocol specifications given via Prolog rules, and enriched via a limited form of term rewriting and narrowing to manage symbolic encryption equations. Similarly to us, verification is performed by means of a symbolic model-checker which relies on a *backward* evaluation procedure which takes as input a set of insecure states. The analyzer needs to be fed with some inductive lemmas by the user, in the same way theorem provers need to be guided by the user during the proof search process.

In [Bla01], the author proposes an optimized specification of security protocols based on an "attacker view" of protocol security, specified by means of Prolog rules, as in [Mea96]. The approach is effective, and has been applied to prove correctness of a number of real protocols. The verification algorithm performs a *backward depth-first* search, which seems to be closely related to our evaluation strategy, and uses an intermediate code optimization using a technique similar to *unfolding*, which we plan to study as future work. On the other hand, we think that the multiset rewriting formalism which we use is more amenable to an automatic translation from the usual protocol notation. As we stressed in Section 11.1, ensuring faithfulness between the intended semantics of a protocol and its specification is necessary to prove correctness. Also, with respect to [Bla01], we use a cleaner treatment for nonces, and we don't have to use approximations (which may introduce false attacks) except for *invariant strengthening*, which can be controlled by the user.

Finally, we mention some works concerning the process of translation from the usual informal notation for protocols, which we plan to study as part of our future work. Existing approaches include `Casper` [Low98], a compiler from protocol specifications into the CSP process algebra, oriented towards verification in FDR, and `CAPSL` [Mil97], a specification

language which can be compiled into an intermediate language and used to feed tools like Maude [DMT98] or the NRL analyzer [Mea96]. Finally, [JRV00] presents an automatic compilation process into rewriting rules which is able to manage infinite-state models.

Summary of the Chapter. *In this chapter we have illustrated the use of our framework based on linear logic for the specification and analysis of authentication protocols. Protocols and security properties can be specified using a uniform logic corresponding to multiset rewriting enhanced with nonce management via universal quantification, and validated by means of a bottom-up evaluation algorithm which performs backward search starting from a protocol/intruder theory and a set of axioms representing security violations.*

We have presented a set of examples taken from classic literature on security protocols, and we have shown that our verification methodology can be used both for finding design errors and for validating protocols with respect to a given intruder theory and a given security property. Unlike most traditional approaches based on model-checking, our verification strategy does not pose any limitation on the number of simultaneous sessions which can be analyzed, and allows any principal to take part into different protocol runs simultaneously, possibly with different roles.

Chapter 12

Conclusions

In this thesis we have a presented a specification language, based on a fragment of Girard's linear logic, which is suitable for the specification of concurrent systems, and in particular of *parameterized* and *infinite-state* systems. Furthermore, we have enriched the language with a novel computational model based on *bottom-up* evaluation of linear logic programs, which has strong connections with symbolic model checking procedures. We have shown different classes of parameterized and infinite-state systems which can be validated using this machinery. Summarizing, we believe that this thesis provides some original contribution at least on the following points:

- Regarding the specification language in itself, we think that a strong point in favour of our approach is the seamless integration of specialized constraint solvers into a substratum consisting of a multiset-rewriting-like logic. Further extensions like, e.g., equational rewriting theories or more complex type theories might be plugged into our system with a limited effort. Finally, universal quantification in goals provides a logical and clean way to specify name generation. We have shown the last point in the *test-and-lock* example of Section 10.5 and in Chapter 11, where the quantifier is used to generate *nonces*;
- we have defined a new *bottom-up* semantics for a fragment of first-order linear logic (with or without constraints). In particular, we have extended the so-called C-semantics of [FLMP93, BGLM94] to a fragment of first-order linear logic including universal quantification in goals. As a result, we have also obtained the extension of the C-semantics to Horn clauses with embedded universal quantification;
- as a result of our semantical investigation, we have obtained *decidability* results for some fragments of *first-order* linear logic (with constraints or without constraints), with applications for the verification of infinite-state systems (as in the case of the *ticket* protocol). We expect further results in this direction from our future work;

- we have presented a verification procedure which can be used either to detect design errors (e.g. *attacks* for authentication protocols) or to validate infinite-state systems. In particular, this procedure is suitable to study a wide class of *safety* properties. Its strength is due to the following reasons: a *symbolic* computation strategy based on first-order *unification* (in the case of the language $\mathrm{LO}_\forall$) or on specialized constraint solvers (in the case of $\mathrm{LO}(\mathcal{C})$); the possibility of expressing safety properties via *upward-closed* sets; a *backward* evaluation strategy (we remark that using a *forward* strategy would not yield the same results); a *subsumption* mechanism (and also *static analysis* techniques) to relieve the state explosion problem;

- finally, we have applied our verification procedure to verify safety properties for a wide class of *parameterized* and *infinite-state systems* ranging from mutual exclusion protocols, to broadcast and authentication protocols. We have also obtained some original results. For instance, we have verified, to our knowledge for the first time, a parameterized formulation, admitting an arbitrary number of clients and servers, of the so-called *ticket* protocol. It is important to note that our specification of the ticket protocol is faithful to its original formulation, in fact we model integer variables using integer variables (which can grow *unboundedly*). In other words, we do not abstract away their values, and we do not use reduction to a *finite-state* problem as in some alternative approaches.

We summarize in Table 12.1 all the experiments that we have carried out in this thesis, using the tool presented in Appendix A. The column 'Section' indicates the section in which the relevant experiment has been presented, a symbol $\surd$ in column 'Static Opt' indicates the use of static analysis techniques like invariant strengthening or pruning (the abstraction α of Definition 9.60 is always used in the case of the ticket algorithm); the column 'Steps' denotes the number of iterations needed to reach a fixpoint (or before finding a security violations); 'Size' is the number of multisets contained in the fixpoint (or at the time the violation was found); finally, 'Time' is the execution time in seconds (not applicable for the readers/writers example which has been verified by hand).

12.1 Future Work

We conclude by outlining the directions of future research we believe more promising.

We are considering enriching our specification language with a type theory, possibly based on dependent types, along the tradition of logical frameworks [Pfe01, CP02]. Dependent types provide more flexible specifications, for instance, in security protocol specifications, they can be used to enforce dependency between an encryption key and its owner [Cer01b]. More importantly, dependent types allow one to represent and manipulate *computations*

Example	Section	Static Opt	Steps	Size	Time	Verified
Producer/Comsumer	7.3.1		13	16	56	*yes*
A Petri Net For Mutual Exclusion	7.3.2		7	14	< 1	*yes*
	7.3.2	$\surd$	1	3	< 1	*yes*
A Petri Net For Mutual Exclusion (Dyn.Gen.)	7.3.2		7	20	< 1	*yes*
	7.3.2	$\surd$	1	3	< 1	*yes*
Readers/Writers	8.4	$\surd$	1	4	*n.a.*	*yes*
Ticket Protocol (Multi-client, Single-server)	9.4		17	201	126	*yes*
	9.4	$\surd$	9	23	< 1	*yes*
Ticket Protocol, Dyn.Gen. (Multi-client, Single-server)	9.4		17	222	150	*yes*
	9.4	$\surd$	10	32	< 1	*yes*
Ticket Protocol, Dyn.Gen. (Multi-client, Multi-server)	9.4	$\surd$	19	141	15	*yes*
Test and Lock	10.5		7	12	< 1	*yes*
Millen's ffgg Protocol	11.4.1		14	306	1335	*attack*
Millen's ffgg Protocol (corrected)	11.4.1		14	27	2419	*yes*
Needham-Schroeder Protocol (strong correctness)	11.4.2		13	294	45	*attack*
Needham-Schroeder Protocol (weak correctness)	11.4.2		13	304	516	*yes*
	11.4.2	$\surd$	12	299	63	*yes*
Corrected Needham-Schroeder (strong correctness)	11.4.3		14	1575	791	*yes*
	11.4.3	$\surd$	9	402	31	*yes*
Otway-Rees Protocol	11.4.4		5	10339	4272	*attack*

Table 12.1: A summary of the experiments carried out in the thesis

as objects. In the context of security protocols, this would amount to representing traces of attacks as terms.

We are currently making further experiments on other classes of protocols, including the field of security protocols. We want to carry out more experiments on different domains (other than integer numbers) and we want to study new abstraction techniques (see Section 9.3) and automatic methods to generate invariants for pruning the search space (see Section 6.1.2). Furthermore, we are investigating the issue of termination of the bottom-up evaluation algorithm for more extended linear logic fragments than the monadic fragments discussed in Section 9.3 and Section 10.4.

We also want to find out whether it is possible to study different classes of problems or different classes of properties (e.g. *liveness* properties) for infinite-state systems. A starting point could be [FS01], where it is shown that the construction of *well-structured transition systems* can be used to deal with the *termination*, *inevitability* and *boundedness* problems. Solution of these problems is based on *tree-saturation methods*, which represent computations by means of a finite *tree-like* structure.

We believe that improving the implementation of our verification tool (see Appendix A) could allow us to analyze more complex protocols than the ones considered so far. In particular, we need to optimize multiset unification (and the related operation of subsumption) and term representation. We are currently studying *ad-hoc* data structures for representing (constrained) multisets, when dealing with specific domains.

From a logic programming perspective, we can extend the results presented in this thesis either by considering more complex linear logic languages [And92, HM94, Mil96, DM01], or more complex observational semantics, like the S-semantics of [FLMP93, BGLM94]. We would also like to extend the decidability results presented in Section 9.3 and Section 10.4. Finally, we would also investigate the connection between our multiset rewriting logic and concurrent constraint programming languages [Sar93], where constraints are used, at the programming level, as a means to provide communication and synchronization primitives between processes running in parallel. In particular, we want to consider the *tccp* language described in [dBGM00], which deals with a timed extension of CCP languages, suitable to specify *reactive* systems.

Bibliography

[ABJN99] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In N. Halbwachs and D. Peled, editors, *Proceedings 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 134–145, Trento, Italy, 1999. Springer-Verlag.

[ACJT96] P. A. Abdulla, K. Cerāns, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proceedings 11th Annual International Symposium on Logic in Computer Science (LICS'96)*, pages 313–321, New Brunswick, New Jersey, 1996. IEEE Computer Society Press.

[AJ98] P. A. Abdulla and B. Jonsson. Verifying Networks of Timed Processes. In B. Steffen, editor, *Proceedings 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 298–312, Lisbon, Portugal, 1998. Springer-Verlag.

[AJ01a] P. A. Abdulla and B. Jonsson. Channel Representations in Protocol Verification. In K. L. Larsen and M. Nielsen, editors, *Proceedings 12th International Conference on Concurrency Theory (CONCUR'2001)*, volume 2154 of *LNCS*, pages 1–15, Aalborg, Denmark, 2001. Springer-Verlag.

[AJ01b] P. A. Abdulla and B. Jonsson. Ensuring Completeness of Symbolic Verification Methods for Infinite-State Systems. *Theoretical Computer Science*, 256(1-2):145–167, 2001.

[ALPT93] J.-M. Andreoli, L. Leth, R. Pareschi, and B. Thomsen. True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings Theory and Practice of Software Development, International Joint Conference CAAP/FASE (TAPSOFT'93)*, volume 668 of *LNCS*, pages 182–198, Orsay, France, 1993. Springer-Verlag.

[AMT94] A. W. Appel, J. S. Mattson, and D. R. Tarditi. A lexical analyzer generator for Standard ML. Version 1.6.0. Department of Computer Science, Princeton University, October 1994. See URL http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Lex/.

[AN00] P. .A. Abdulla and A. Nylén. Better is Better than Well: On Efficient Verification of Infinite-State Systems. In *Proceedings 15th Annual International Symposium on Logic in Computer Science (LICS'00)*, pages 132–140, Santa Barbara, California, 2000. IEEE Computer Society Press.

[AN01] P. .A. Abdulla and A. Nylén. Timed Petri Nets and BQOs. In *Proceedings 22nd International Conference on Application and Theory of Petri Nets (ICATPN'01)*, Newcastle upon Tyne, U.K., 2001.

[And92] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[And96] J.-M. Andreoli. Coordination in LO. In J.-M. Andreoli, C. Hankin, and D. Le Metayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 42–64. Imperial College Press, London, UK, 1996.

[AP91a] J.-M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In A. Paepck, editor, *Proceedings 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, pages 212–229, Phoenix, Arizona, 1991. ACM Press.

[AP91b] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[APC97] J.-M. Andreoli, R. Pareschi, and T. Castagnetti. Static Analysis of Linear Logic Programming. *New Generation Computing*, 15(4):449–481, 1997.

[APR+01] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. D. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 221–234, Paris, France, 2001. Springer-Verlag.

[Apt90] K. R. Apt. *Logic Programming*, chapter 10, pages 493–574. Handbook of Theoretical Computer Science. Elsevier Science, 1990.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, Digital Equipment Corporation Systems Research Center, 1989.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In S. Graf and M. I. Schwartzbach, editors, *Proceedings 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 188–203, Berlin, Germany, 2000. Springer-Verlag.

[BD02] M. Bozzano and G. Delzanno. Beyond Parameterized Verification. In J.-P. Katoen and P. Stevens, editors, *Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 221–235, Grenoble, France, 2002. Springer-Verlag.

[BDLM00] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. Object Calculi in Linear Logic. *Journal of Logic and Computation*, 10(1):75–104, 2000.

[BDM97] M. Bozzano, G. Delzanno, and M. Martelli. A Linear Logic Specification of Chimera. In *Proceedings Workshop on Transactions and Change in Logic Databases (DYNAMICS'97)*, Port Jefferson, Long Island, New York, 1997.

[BDM+99] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic Programming and Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In K.R. Apt, V.W. Marek, M. Truszczynski, and D.S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, Springer Series in Artificial Intelligence, pages 5–32. Springer-Verlag, 1999.

[BDM00] M. Bozzano, G. Delzanno, and M. Martelli. A Bottom-up Semantics for Linear Logic Programs. In *Proceedings 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 92–102, Montreal, Canada, 2000. ACM Press.

[BDM01a] M. Bozzano, G. Delzanno, and M. Martelli. An Effective Bottom-Up Semantics for First Order Linear Logic Programs. In H. Kuchen and K. Ueda, editors, *Proceedings 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 138–152, Tokyo, Japan, 2001. Springer-Verlag.

[BDM01b] M. Bozzano, G. Delzanno, and M. Martelli. On the Relations between Disjunctive and Linear Logic Programming. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming - Selected Papers from AGP 2000*, volume 48 of *ENTCS*, La Habana, Cuba, 2001.

[BDM02] M. Bozzano, G. Delzanno, and M. Martelli. An Effective Fixpoint Semantics for Linear Logic Programs. *Theory and Practice of Logic Programming*, 2(1):85–122, January 2002.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetics. In O. Grumberg, editor, *Proceedings 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411, Haifa, Israel, 1997. Springer-Verlag.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418, Chicago, Illinois, 2000. Springer-Verlag.

[BKK+96] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings 1st International Workshop on Rewriting Logic and its Applications*, volume 4 of *ENTCS*, Asilomar Conference Center, Pacific Grove, California, 1996.

[Bla01] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of 14th Computer Security Foundation Workshop (CSFW'01)*, pages 82–96, Cape Breton, Nova Scotia, Canada, 2001.

[Boz01] M. Bozzano. Ensuring Security through Model Checking in a Logical Environment (Preliminary Results). In *Proceedings of Workshop on Specification, Analysis and Validation for Emerging Technologies in Computational Logic (SAVE'01)*, Paphos,Cyprus, December 2001.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fix-Points. In *Proceedings 4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[CD97] R. Di Cosmo and V. Danos. The Linear Logic Primer, 1997. Notes for an introductory course on linear logic. Current revision available from URL http://www.pps.jussieu.fr/∼dicosmo/CourseNotes/LinLog/.

[CDKS00] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting Strands in Linear Logic. In E. Clarke H. Veith, N. Heintze, editor, *Proceedings 2000 Workshop on Formal Methods and Computer Security (FMCS'00)*, Chicago, Illinois, 2000.

[CDL+99] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchell, and A. Scedrov. A Meta-notation for Protocol Analysis. In R. Gorrieri, editor, *12th Computer Security Foundations Workshop (CSFW'99)*, pages 55–69, Mordano, Italy, 1999.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings 1st International Workshop on Rewriting Logic and its Applications*, volume 4 of *ENTCS*, Asilomar Conference Center, Pacific Grove, California, 1996.

[Cer94] I. Cervesato. Petri Nets as Multiset Rewriting Systems in a Linear Framework, 1994. Unpublished manuscript. Draft available from URL http://theory.stanford.edu/∼iliano/forthcoming.html.

[Cer95] I. Cervesato. Petri Nets and Linear Logic: a Case Study for Logic Programming. In M. Alpuente and M. I. Sessa, editor, *Proceedings 1995 Joint Conference on Declarative Programming (GULP-PRODE'95)*, pages 313–318, Marina di Vietri, Italy, 1995. Palladio Press.

[Cer01a] I. Cervesato. The Dolev-Yao Intruder is the Most Powerful Attacker. In J. Halpern, editor, *Proceedings 16th Annual International Symposium on Logic in Computer Science (LICS'01)*, Boston, Massachusetts, 2001. Short paper.

[Cer01b] I. Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. In A. Seda, editor, *Proceedings 1st Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT'00)*, Cork, Ireland, 2001.

[CFGR95] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):47–68, 1995.

[Chi95] J. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[Cir01] H. Cirstea. Specifying Authentication Protocols Using Rewriting and Strategies. In I. V. Ramakrishnan, editor, *Proceedings of Conference on Practical*

Aspects of Declarative Languages (PADL'01), volume 1990 of *LNCS*, pages 138–152, Las Vegas, Nevada, 2001. Springer-Verlag.

[CJ97] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature, 1997. Web Draft Version 1.0 available from `http://www-users.cs.york.ac.uk/~jac/`.

[CP02] I. Cervesato and F. Pfenning. A Linear Logical Framework, 2002. To appear in *Information and Computation*.

[dBGM00] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[Del00] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In E. A. Emerson and A. P. Sistla, editors, *Proceedings 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, Chicago, Illinois, 2000. Springer-Verlag.

[Del01] G. Delzanno. Specifying and Debugging Security Protocols via Hereditary Harrop Formulas and λProlog - A Case-study -. In H. Kuchen and K. Ueda, editors, *Proceedings 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 123–137, Tokyo, Japan, 2001. Springer-Verlag.

[DEP99] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proceedings 1999 Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 50–66, Madrid, Spain, 1999. Springer-Verlag.

[DES76] Federal Information Processing Standard 46 - the Data Encryption Standard, 1976.

[Dic13] L. E. Dickson. Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors. *American Journal of Mathematics*, 35:413–422, 1913.

[DM01] G. Delzanno and M. Martelli. Proofs as Computations in Linear Logic. *Theoretical Computer Science*, 258(1-2):269–297, 2001.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, 1998.

[DP99] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 223–239, Amsterdam, The Netherlands, 1999. Springer-Verlag.

[DP01] N. Dershowitz and D. Plaisted. Rewriting. In A.Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 1. Elsevier Science, 2001.

[DRB01] G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 298–310, Paris, France, 2001. Springer-Verlag.

[DY83] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[Ede85] E. Eder. Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proceedings 14th International Symposium on Logic in Computer Science (LICS'99)*, pages 352–359, Trento, Italy, 1999. IEEE Computer Society Press.

[EN94] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets - a Survey. *Journal of Informatik Processing and Cybernetics*, 30(3):143–160, 1994.

[EN95] E. Emerson and K. Namjoshi. Reasoning about Rings. In *Proceedings 22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 85–94, S.Francisco, California, 1995. ACM Press.

[EN98] E.A. Emerson and K.S. Namjoshi. On Model Checking for Non-Deterministic Infinite-State Systems. In *Proceedings 13th International Symposium on Logic in Computer Science (LICS'98)*, pages 70–80, Indianapolis, Indiana, 1998. IEEE Computer Society Press.

[EP91] C. Elliott and F. Pfenning. A Semi-Functional Implementation of a Higher-Order Logic Programming Language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[EW90] U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *Proceedings of Colloquium on Trees in Algebra and Programming*, volume 389 of *LNCS*, pages 147–161, Copenhagen, Denmark, 1990. Springer-Verlag.

[Fin87] A. Finkel. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In T. Ottmann, editor, *Proceedings 14th International Colloquium on Automata, Languages and Programming (ICALP'87)*, volume 267 of *LNCS*, pages 499–508, Karlsruhe, Germany, 1987. Springer-Verlag.

[FLMP93] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.

[FO97] L. Fribourg and H. Olsén. Reachability Sets of Parametrized Rings As Regular Languages. In F. Moller, editor, *Proceedings 2nd International Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *ENTCS*, Bologna, Italy, 1997. Elsevier Science.

[FPP01] F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of Sets of Infinite State Processes Using Program Transformation. In A. Pettorossi, editor, *Proceedings 11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, Paphos, Cyprus, 2001.

[Fri99] L. Fribourg. Constraint Logic Programming Applied to Model Checking. In A. Bossi, editor, *Proceedings 9th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *LNCS*, pages 30–41, Venezia, Italy, 1999. Springer-Verlag.

[FS01] A. Finkel and P. Schnoebelen. Well-Structured Transition Systems Everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[Gal86] J.H. Gallier. *Logic for Computer Science*. Harper and Row, 1986.

[GDL95] M. Gabbrielli, M. G. Dore, and G. Levi. Observable semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[GM93] M. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[GS92] S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GZ98] E. P. Gribomont and G. Zenner. Automated Verification of Szymanski 's Algorithm. In B. Steffen, editor, *Proceedings 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 424–438, Lisbon, Portugal, 1998. Springer-Verlag.

[Har01] R. Harper. Programming in Standard ML, 2001. Lecture Notes, Carnegie Mellon University. Draft available from URL http://www-2.cs.cmu.edu/∼rwh/smlbook/.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings London Mathematical Society*, 2:326–336, 1952.

[HJJ+95] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Proceedings 1st International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *LNCS*, pages 89–110, Aarhus, Denmark, 1995. Springer-Verlag.

[HM94] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[Hol88] G. J. Holzmann. Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, 69(2):32–44, 1988.

[HP94] J. Harland and D. J. Pym. A Uniform Proof-Theoretic Investigation of Linear Logic Programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

[HW98] J. Harland and M. Winikoff. Making Logic Programs Reactive. In *Proceedings Workshop on Transactions and Change in Logic Databases (Dynamics'98)*, pages 43–58, Manchester, UK, 1998.

[ID96] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. A. Henzinger, editors, *Proceedings 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 147–158, New Brunswick, New Jersey, 1996. Springer-Verlag.

[Jen97] K. Jensen. *Coloured Petri-Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, 2 and 3.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[JL87] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings 14th Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119, Munich, Germany, 1987. ACM Press.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[JN00] B. Jonsson and M. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In S. Graf and M. I. Schwartzbach, editors, *Proceedings 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 220–234, Berlin, Germany, 2000. Springer-Verlag.

[JRV00] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'00)*, volume 1955 of *LNCS*, pages 131–160, Reunion Island, France, 2000. Springer-Verlag.

[KM69] R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[KM95] R. Kurshan and K. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KMM+97a] P. Kelb, T. Margaria, M. Mendler, , and C. Gsottberger. MOSEL: A flexible toolset for monadic second-order logic. In E. Brinksma, editor, *Proceedings 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *LNCS*, pages 183–202, Enschede, The Netherlands, April 1997. Springer-Verlag.

[KMM+97b] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 424–435, Haifa, Israel, 1997. Springer-Verlag.

[Kop95] A. P. Kopylov. Decidability of Linear Affine Logic. In D. Kozen, editor, *Proceedings 10th Annual International Symposium on Logic in Computer Science (LICS'95)*, pages 496–504, San Diego, California, 1995. IEEE Computer Society Press.

[KY95] N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 7(2):113–149, 1995.

[Laf95] Y. Lafont. From Proof Nets to Interaction Nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 225–247. Cambridge University Press, 1995.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, , and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proceedings 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 346–357, Paris, France, 1997. ACM Press.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LMM88] J.-L. Lassez, J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[Low95] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In T. Margaria and B. Steffen, editors, *Proceedings 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166, Passau, Germany, 1996. Springer-Verlag.

[Low98] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[LS97] D. Lesens and H. Saidi. Abstraction of Parameterized Networks. In F. Moller, editor, *Proceedings 2nd International Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *ENTCS*. Elsevier Science, Bologna, Italy, 1997.

[Mah92] M. Maher. A CLP View of Logic Programming. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, volume 632 of *LNCS*, pages 364–383, Volterra, Italy, 1992. Springer-Verlag.

[Mar99] A. Marcone. Fine and Axiomatic Analysis of the quasi-orderings on $\mathcal{P}(q)$. Technical Report Technical Report 17/99/RR, Dipartimento di Matematica e Informatica dell'Università di Udine, 1999.

[McD97] R. McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, 1997.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. Model Checking for Security Protocols. Technical Report CMU-CS-97-139, School of Computer Science, Carnegie Mellon University, 1997.

[Mea96] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil85] E. C. Milner. Basic wqo- and bqo-theory. In I. Rival, editor, *Graphs and Orders*, pages 487–502. D. Reidel Publishing Company, 1985.

[Mil92] D. Miller. The π-Calculus as a Theory in Linear Logic: Preliminary Results. In E. Lamma and P .Mello, editors, *Proceedings Workshop on Extensions of Logic Programming*, volume 660 of *LNCS*, pages 242–265, Bologna, Italy, 1992. Springer-Verlag.

[Mil96] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[Mil97] J. K. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[Mil99] J. K. Millen. A Necessarily Parallel Attack. In *Proceedings Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[MMP96] R. McDowell, D. Miller, and C. Palamidessi. Encoding Transition Systems in Sequent Calculus. In J.-Y. Girard, M. Okada, and A. Scedrov, editors, *Proceedings Linear Logic 96 Tokyo Meeting*, volume 3 of *ENTCS*, Keio University, Tokyo, Japan, 1996.

[MNPS91] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[MQS00] K. McMillan, S. Qadeer, and J. Saxe. Induction and compositional model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 312–327, Chicago, Illinois, 2000. Springer-Verlag.

[MRL91] J. Minker, A. Rajasekar, and J. Lobo. Theory of Disjunctive Logic Programs. In J.L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson*, pages 613–639. MIT Press, 1991.

[Nil00] M. Nilsson. *Regular Model Checking*. PhD thesis, Department of Information Technology, Uppsala University, 2000.

[NS78] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[OR87] D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T. A. Henzinger, editors, *Proceedings 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, New Jersey, 1996. Springer-Verlag.

[Pal90] C. Palamidessi. Algebraic properties of idempotent substitutions. In M. S. Paterson, editor, *Proceedings 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *LNCS*, pages 386–399, Warwick University, England, 1990. Springer Verlag.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pau96] L. C. Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[Pfe01] F. Pfenning. Logical Frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001.

[PRZ01] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic Deductive Verification with Invisible Invariants. In T. Margaria and W. Yi, editors, *Proceedings 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 82–97, Genova, Italy, 2001. Springer-Verlag.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNCS*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In E. A. Emerson and A. P. Sistla, editors, *Proceedings 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 328–343, Chicago, Illinois, 2000. Springer-Verlag.

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999.

[Rei85] W. Reisig. *Petri Nets, an introduction*. EATCS, Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RV95] M. Rusinowitch and L. Vigneron. Automated Deduction with Associative and Commutative Operators. *Applicable Algebra in Engineering, Communication and Computing*, 6:23–56, 1995.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[SMC00] P. Syverson, C. Meadows, and I. Cervesato. Dolev-Yao is no better than Machiavelli. In P. Degano, editor, *Proceedings of the First Workshop on Issues in the Theory of Security (WITS'00)*, pages 87–92, Geneva, Switzerland, 2000.

[STC98] M. Silva, E. Teruel, and J. Colom. Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems. In G. Rozenberg and W. Reisig, editors, *Lectures in Petri Nets. I: Basic Models*, volume 1491 of *LNCS*, pages 309–373. Springer-Verlag, 1998.

[TA00] D. R. Tarditi and A. W. Appel. ML-Yacc User's Manual. Version 2.4. Microsoft Research and Department of Computer Science, Princeton University, April 2000. See URL http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/.

[Tho90] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 134–191. Elsevier Science, 1990.

[Tro92] A.S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Vig95] L. Vigneron. Positive Deduction modulo Regular Theories. In H. K. Büning, editor, *Proceedings 1995 Annual Conference of the European Association for Computer Science Logic (CSL'95)*, volume 1092 of *LNCS*, pages 468–485, Paderborn, Germany, 1995. Springer-Verlag.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In J. Sifakis, editor, *Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80, Grenoble, France, 1989. Springer-Verlag.

Appendix A

Verification Tool

In order to run the experiments described in chapters 7 through 11, we have built a (prototypical) verification tool implementing the backward reachability algorithm for the *covering* problem discussed in Chapter 6. In this section we give a brief overview about the use and implementation of the tool.

A.1 General Description

The verification tool can be viewed as an implementation of the (semi)-algorithm for the *covering* problem presented in Figure 6.3. In particular, it supports computation of the bottom-up semantics for first-order LO programs with clauses built upon the logical connectives ⅋, $\circ\!\!-$, $\top$, and the universal quantifier $\forall$ in goals. The verification algorithm exploits the *upward-closure* property of LO axioms of the form $H \circ\!\!- \top$ and is therefore tailored to solve the so-called *covering* problem (see Section 6.1.1). For this reason, the logical constant **1** (and consequently the multiplicative conjunction $\otimes$) are not supported. Furthermore, the additive conjunction & is currently not supported (though it could be added with a limited effort). In order to run the experiments described in Chapter 9, a (core of a) specialized constraint solver has been implemented. In particular, the tool supports the DC-constraints of Chapter 9.

The algorithm evaluates the bottom-up semantics of an LO program, producing in output the backward reachability set computed at every iteration. The fixpoint is computed (modulo *state-explosion*) any time termination is guaranteed. In case an *attack* (i.e., a counterexample violating the property under investigation) is found, the corresponding execution trace can be recovered and examined.

In Section A.3 we give more details about how the algorithm works.

A.2 Implementation Notes

Following [EP91], the verification tool has been implemented in Standard ML (see [Pau96] or [Har01] for a tutorial). We chose SML because it provides high-level programming features together with a reasonable computational efficiency.

The verification tool has been built as a collection of independent modules implementing, respectively: language parsing, term management, a specialized constraint solver for a subset of linear constraints, (constrained) multisets, the bottom-up evaluation procedure, and a user frontend. A parser supporting the languages $\mathrm{LO}_\forall$ and $\mathrm{LO}(\mathcal{C})$ has been produced using ML-Lex [AMT94] and ML-Yacc [TA00], a lexical analyzer and an automatic parser generator available for the SML environment. Some parts of the code dealing with term management, e.g. unification, have been adapted from the source code for an interpreter for the language Lolli [HM94], originally written by J. Hodas.

A.3 How It Works

The verification tool takes as input a given system specification as a set of LO clauses and a set of LO axioms representing the set of final states, a specification of the set of initial states, and possibly some pruning invariants (see Section 6.1.2.2). The output consists of the sets of configurations computed at each iteration and the resulting fixpoint, if the computation is terminating. More details are given below.

Initial States. The tool allows one to give a *partial* specification of the set of the initial states. It is possible to require a certain atom (or multiset of atoms) to belong to every initial state and/or impose an atom (multiset of atoms) not to belong to any initial state. The verification algorithm signals it has *possibly* found an *attack* every time it finds a state which is compatible with the partial specification and the user must take care of checking if it is really an attack or not. Exact specification of the set of initial states is also possible. However, partial specification is more flexible because it may help one to find additional hypotheses under which a security violation might take place.

Invariant Strengthening. The tool supports the invariant strengthening technique (see Section 6.1.2.1, consisting in enriching the set of the states the bottom-up evaluation starts from (i.e., enriching the set of logical axioms). We have also plugged the abstraction function α of Section 9.3, which can be set to validate specifications written in LO enriched with a subclass of linear integer constraints.

```
Procedure Symbolic Fixpoint
input P: a set of LO clauses
       Ax: a set of LO axioms
       Initial: a set of initial states
       Invar: pruning invariants
output F_sym(P)
begin
    Step := 1
    Interp := S_Ax(∅); Last := Interp
    repeat
        Step := Step + 1
        Last := S_P(Last)
        Last := prune(Last, Invar)
        (Interp, Last) := cut_subsumed(Interp, Last)
        print Step, Interp
        if Last ∩ Initial ≠ ∅ then print "attack found"
    until Last = ∅
    print "fixpoint"
end
```

Figure A.1: Pseudo-algorithm for bottom-up evaluation

Pruning. The tool supports the pruning methodology (see Section 6.1.2.2). Specifically, it is possible to specify *pruning invariants* in the form of *upward-closed* sets of states which are known to be unreachable. Upward-closed sets of states can be specified as LO axioms. The tool checks every computed configuration against the pruning invariants, and possibly cuts the search space accordingly. At the moment, pruning invariants must be provided by the user and are not tested for correctness (see also Section A.5).

Subsumption. At every iteration of the fixpoint computation, subsumption checks are performed in order to discard redundant information. Intuitively, an element (*constraint* in the terminology of [AJ01b]) is redundant if its *denotation* is contained in the denotation of some previously computed element. Subsumption checks work as follows. Every newly generated element is checked against the already computed ones. If it is subsumed, then it can be immediately discarded, otherwise it is inserted. Elements which become redundant as a result of this insertion are in turn discarded.

Evaluation Algorithm. The bottom-up evaluation algorithm, written in pseudo-code, in presented in Figure A.1. At any given iteration, the variable *Interp* contains the current (global) interpretation computed so far, while the variable *Last* contains the subset

of elements of *Interp* computed during the last step. The subroutine *prune* prunes a given interpretation with respect to the given set of pruning invariants. The subroutine *cut_subsumed* takes *Interp* and *Last* as input and inserts elements of *Last* into *Interp* performing the subsumption checks described in the previous paragraph (subsumed elements are removed from *Last*).

A.4 Experimental Environment

In Chapters 7 through 11 we have presented some experiments which we have carried out using our verification tool. All experiments have been executed on a Pentium III 450 Mhz, under Linux 2.2.13-0.9, and running Standard ML of New Jersey, Version 110.0.7.

A.5 Future Work

We have singled out a number of improvements of the current prototype which we want to carry out. On the theoretical level, we plan to enrich our tool with support for (automatic or semi-automatic) generation of invariants (see Section 6.1.2.1) and cut conditions (see 6.1.2.2). To this aim, we plan to integrate a module for automatic generation of structural invariants for Petri nets [CFGR95]. We also plan to interface our tool with general constraint solvers for different domains.

From an implementation point of view, some improvements are needed in order to get a better performance. In particular, the critical operation is multiset unification (directly implemented in SML), which requires to consider (classical) unification between sub-multisets of terms of the original multisets. The most expensive phase of the verification algorithm is by far the *subsumption* check, which makes heavy use of multiset unification. Subsumption is indeed the basis of the verification algorithm's power. Future work includes implementing some kind of *indexing* to order terms inside multisets, so to make unification faster, and optimizing the subsumption phase (e.g. using *heuristics*). We also need to optimize the term representation, which is currently very naive.