FACULTÉ DES SCIENCES
Département d'Informatique

UNIVERSITÉ LIBRE DE BRUXELLES,
UNIVERSITÉ D'EUROPE
ULB

Development and Validation of Distributed Reactive Control Systems

Cédric MEUTER

Dissertation présentée en vue de l'obtention
du grade de **Docteur en Sciences**

Année académique 2007-2008

PÔLE UNIVERSITAIRE
EUROPÉEN
DE BRUXELLES
WALLONIE

ACADÉMIE UNIVERSITAIRE
WALLONIE-BRUXELLES

This dissertation has been written under the supervision of Prof. Thierry Massart (Université Libre de Bruxelles, Belgique). The members of the Jury are:

- Prof. Raymond Devillers (Université Libre de Bruxelles, Belgique)
- Dr. Loïc Hélouët (IRISA Rennes, France)
- Prof. Michaël Leuschel (Heinrich-Heine-Universität Düsserldorf, Deutschland)
- Prof. Jean-François Raskin (Université Libre de Bruxelles, Belgique)
- Dr. Laurent Van Begin (Université Libre de Bruxelles, Belgique)

Abstract

A *reactive* control system is a computer system reacting to certain stimuli emitted by its environment in order to maintain it in a desired state. *Distributed* reactive control systems are generally composed of several processes, running in parallel on one or more computers, communicating with one another to perform the required control task. By their very nature, *distributed reactive control systems* are hard to design. Their distributed nature and/or the communication scheme used can introduce subtle unforeseen behaviours. When dealing with critical applications, such as plane control systems, or traffic light control systems, those unintended behaviours can have disastrous consequences. It is therefore essential, for the designer, to ensure that this does not happen. For that purpose, rigorous and systematic techniques can (and should) be applied as early as possible in the development process. In that spirit, this work aims at providing the designer with the necessary tools in order to facilitate the *development* and *validation* of such distributed reactive control systems. In particular, we show how using a dedicated language called dSL (Distributed Supervision language) can be used to ease the development process. We also study how validations techniques such as *model-checking* and *testing* can be applied in this context.

Résumé

Un système de contrôle *réactif* est un système informatique réagissant a des stimuli de son environnement de façon à maintenir ce dernier dans un état désiré. Les systèmes de contrôle réactifs *distribués* sont en général composés de plusieurs processus s'exécutant en parallèle sur une ou plusieurs plateforme(s), et communiquent entre eux, de manière à réaliser la tâche de contôle requise. Par nature, les *systèmes de contrôle réactifs distribués* sont difficiles à concevoir. Leur nature distribuée et/ou les méchanismes de communication utilisés peuvent introduire des comportements étranges et inattendus. Dans le cadre d'applications critiques, telle que des systèmes de contrôle aéronautique ou des systèmes de contrôle de feux de signalisation, ces comportements inattendus peuvent avoir des conséquences désastreuses. Il est dès lors primordial, pour le concepteur de tels systèmes de s'assurer que ceci ne se produise pas. A ces fins, des techniques rigoureuses et systématiques peuvent (et devraient) être utilisées aussitôt que possible dans le processus de développement. Dans cette optique, ce travail tend à fournir au concepteur les outils nécessaires pour faciliter le développement et la validation de tels systèmes de contrôle réactif distribués. En particulier, nous montrons comment l'utilisation d'un language de programmation dédié, appelé dSL (Distributed Supervision Language) peut être utilisé pour faciliter le processus de développement. Nous étudions également comment des techniques de validation formelles telles que le *model-cheking* et le *test* peuvent être appliquée dans ce contexte.

Thank You!

A thesis is a long journey. One that requires determination, perseverance, and of course a lot of hard work. But most of all, it requires a lot of help and guidance from those around you, on a scientific as well as personal level. I would like to take the opportunity that is given to me in these few pages to thank them in due form.

A little more than five years ago, at a barbecue, one of my professors at the time, Thierry Massart, handed me a sausage and asked if I wanted to join in the wonderful world of academic research. Of course, at the time, I did not realise the many challenges that this decision would entail. But anyway, I signed on for what would be one of the most rewarding experience of my life. For this immense opportunity, for being my adviser ever since, and for the sausage of course, I am extremely grateful to him.

I would also like to thank my other co-authors, and friends: Bram, Alex, Laurent and Gabriel, without which this dissertation would not have seen the light of day. Working with all of them during these 5+ years was a great pleasure.

About a year before the aforementioned barbecue, I had the privilege of working during my last year as an undergraduate student, at the *Institut de Recherche en Communication et Cybernétique de Nantes* under the supervision of Dr. Franck Cassez. He always managed to find time to guide the overly enthusiastic student that I was back then. For putting me on the right tracks, and for all the french fries jokes, I would like to thank him. Of course, these first few steps would not have been possible without the help of Prof. Jean-François Raskin who accepted to promote my Master's thesis at the time, and my DEA thesis afterwards. I would like express my gratitude to him for making all of this a reality, and also for having kindly accepted to be a member of the jury.

I would also like to thank the other members of the jury: Prof. Raymond Devillers, Dr. Loïc Hélouët, and Prof. Michael Leuschel, for having taken the time to review this dissertation. A special credit has to be awarded to Prof. Devillers for the time and care he put in the meticulous proofreading of this document, which indubitably improved its overall quality.

On a personal note, it was a great pleasure to work at the Computer Science Department during all those years. Many thanks are due to everyone there, for making it such a wonderful working environment. I would like to mention a few of them in particular (in no particular order): Pierre, Eythan, Martin, Gilles, Fred S., Nico M., Seb, Joël, Olivier, Fred P., Nicolás G.D.C., and Manu.

In addition, I would like credit the my *pourristes* friends from Tubize: Chris & Vanina, Alain, David, Manu, Pat & Cindy, and my "adoptive family" in Brussels: Nico & Souk, Nico & Marie. Many thanks to all of you for the good times, the laughs, the trips, and so much more. You continuously remind me that there is so much more to life than computer science, even though a lot of you are computer scientists. Your friendship means the world to me!

And last, but most certainly not least, I would like to thank my parents for their endless love and support. Thank you for everything!

Bruxelles
February 26$^{\text{th}}$, 2008
Cédric Meuter

Contents

Introduction 1
Distributed Reactive Control Systems 2
Development Environment 3
Model Checking 4
Testing 5
Structure of this Thesis 7

1 Preliminaries 11
1.1 Sets, Relations and Functions 11
1.2 Lattice Theory 13
1.2.1 Partially Ordered Set 13
1.2.2 Lattice 14
1.2.3 Fixed Points 16
1.3 Boolean Logics 18
1.3.1 Propositional Boolean Logic 18
1.3.2 Quantified Boolean Logic 19
1.4 Languages and Automata 20
1.5 Model Checking 22
1.5.1 Kripke Structures 22
1.5.2 Linear Temporal Logic 27
1.5.3 Computational Tree Logic 29

2 Distributed Supervision Language 33
2.1 Syntax 35
2.1.1 A Running Example 36
2.1.2 Types 36
2.1.3 Global Variables 38
2.1.4 Sites 39

2.1.5 Methods 39
2.1.6 Events 40
2.1.7 Sequences 44
2.2 Distribution 45
2.2.1 Event-Driven Code 45
2.2.2 Sequential Code 47
2.3 Related Works 48

3 Model Checking dSL Programs 51
3.1 Modeling 52
3.1.1 Modeling a Program 52
3.1.2 Modeling a Distribution 55
3.1.3 Semantics of a Program 60
3.1.4 History 69
3.2 Specification 70
3.3 Verification 74
3.3.1 Communicating Finite State Machine 75
3.3.2 Undecidability Result 76
3.3.3 Constraining the Semantics 80
3.3.4 Verification in Practice 83

4 Testing dSL Programs 87
4.1 Instrumentation 89
4.1.1 Message Passing Programs 90
4.1.2 Shared Variable Programs 92
4.2 Partially Ordered Trace 94
4.2.1 Definition 94
4.2.2 Semantics 95
4.3 Specification 98
4.3.1 Non Temporal Properties 98
4.3.2 LTL Properties 100
4.3.3 CTL Properties 104
4.4 Discussions 109

5 Predicate Detection 111
5.1 Classes of Predicates 112

5.1.1 Local Predicates ... 114
5.1.2 Disjunctive Predicates ... 115
5.1.3 Stable Predicates ... 116
5.1.4 Observer-Independent Predicates ... 118
5.1.5 Conjunctive Predicates ... 120
5.1.6 Linear Predicates ... 123
5.1.7 Regular Predicates ... 124
5.2 A Syntactical Perspective ... 134
5.3 Summary ... 138

6 LTL Trace Checking ... 139
6.1 Automata-Based Approach ... 140
6.1.1 Monitor ... 140
6.1.2 Composition ... 145
6.1.3 Explicit Trace Checking Algorithm ... 147
6.2 Monitor Driven Approach ... 148
6.2.1 Determined Monitors ... 148
6.2.2 Monitor Driven Composition ... 154
6.2.3 Symbolic Trace Checking Algorithm ... 159
6.3 Experimental Results ... 162
6.4 Related Works ... 164

7 CTL Trace Checking ... 167
7.1 A Regular Fragment of CTL ... 168
7.1.1 Slicing Algorithm for EF ... 170
7.1.2 Slicing Algorithm for AG ... 170
7.1.3 Slicing Algorithm for EG ... 172
7.1.4 Trace Checking Algorithm ... 174
7.2 Tuple Based Approach ... 176
7.2.1 Tuple Representation ... 176
7.2.2 Tautology ... 178
7.2.3 Propositions ... 180
7.2.4 Boolean Operators ... 184
7.2.5 Temporal Modalities ... 185
7.2.6 Trace Checking Algorithm ... 188
7.3 Symbolic Representation ... 190

7.4 Experimental results 193

8 Case Study: A Canal Lock Controller 195
8.1 Problem Statement 196
8.2 Designing the Controller 196
8.3 Model Checking the Controller 199
8.4 Testing the Controller 203

Conclusion 207
Personal Contribution 208
Future works 209

A Grammars of dSL 211
A.1 Full Grammar 211
A.2 Simplified Grammar 215

B One-Split Simulation Lemma 219
B.1 Lemma Statement 219
B.2 Preliminary Results 219
B.3 Proof of the Lemma 221

C dSL to Promela: an Example 235
C.1 The dSL Program 235
C.2 The Corresponding Promela Code 236

D The Canal Locks Controllers 241

List of Definitions 247

List of Theorems 251

List of Algorithms 253

List of Figures 255

Bibliography 259

Index 273

« *Okay, brain. You don't like me, and I don't like you, but let's get through this thing and then I can continue killing you with beer.* »

Homer Simpson, *The Simpsons*

Introduction

« *Computers are useless. They can only give you answers.* »

Pablo Picasso

COMPUTERS play a very important role in our daily lives. Most equipments all around us, from cars to coffee machines, toys to nuclear power plants are controlled using some sort of computer system. Computers have also become essential in almost all sectors of our service-oriented economy. A huge number of activity domains rely on them: banks, administration offices, post offices, travel agencies, telecommunications, etc. In essence, computer systems have become a deep-rooted part of our existence, without which we would most certainly be at a loss. However, computers have an inherent flaw. They are only capable of executing programs written by *humans beings*, and are as a consequence subject to *human errors*. Our recent history is filled with so-called "computer errors". Let us mention a few of the most famous *bugs*, taken from [Fleury, 2002].

- On July 4$^{\text{th}}$, 1997, the *Mars Pathfinder* project was put on hold for several days. One of the main goals of the project was to allow a mobile probe, called *Sojourner* to land and collect data from the surface of Mars. Unfortunately, after a couple of days of exploration, the *Sojourner* probe started to re-initialize itself randomly. The error was caused by a problem in the real-time kernel of the probe whereby two tasks were waiting for each other to complete, thus resulting in a *deadlock* situation. The liveness monitor embedded in the system was therefore forced to re-initialize the probe. After a lot of efforts in isolating the source of the error, by recreating the conditions in which the *bug* occurred, the *Jet Propulsion Laboratory* engineers at NASA were able to solve the problem.
- On November 21$^{\text{st}}$, 1985, a computer malfunction at the *Bank of New York* brought the *Treasury Bond Market* to a standstill. The error happened quickly after an upgraded version of the Treasury's software responsible for handling the transactions with Wall Street was loaded. This *bug* was due to an overflow in an

integer variable. As it happened, in the new version of the software, the variable used to store the number of transactions was encoded as a 16 bits integer, while in the previous version this variable was encoded on 32 bits. When the 32.768^{th} transaction request arrived, the variable was reset to 0, as could be expected. The *Bank of New York* had to borrow a record $20 billion from the *Federal Reserve* to compensate for the loss caused by this error.

- From 1985, a malfunction in a radiation therapy device called the *Therac-25* caused the death of 4 patients, and other severe injuries. This malfunction was caused by a *race condition* in the control software. Because of this error, a fast succession of operations could give rise to the delivery of a massive dose of radiations. After 6 major incidents spread over 2 years (!), the decision to stop using the *Therac-25* was finally taken in 1987.

These are only but a few examples, illustrating that a "computer error" can have disastrous consequences, both from a human and economical viewpoint. That is why it is imperative that critical computer systems are thoroughly validated. For that purpose, rigorous and systematical techniques can (and should) be applied, as early as possible in the development process.

In this work, we place ourselves in the context of *distributed reactive control systems*. In particular, we will focus on their *development* and *validation*. We will examine how a dedicated language called dSL (Distributed Supervision Language) can be used to ease their development, and how *model-checking* and *testing* can be used for their validation.

Distributed Reactive Control Systems

A *reactive* control system is a computer system reacting to certain stimuli emitted by the environment it controls in order to maintain it in a desired state. For instance, a temperature control system will react to a signal indicating that the temperature is below a certain threshold by turning on a heater, thus ensuring that the temperature is always adequate. For simple tasks like these, a *centralized* system can do the job. However, when dealing with real-size industrial systems, it is often not the case. If the task is too complex, the designer may choose to decompose it into several sub-tasks to be performed in parallel. Also, the equipment may be geographically distributed, in which case, the designer will be forced to distribute the control system over several computer platforms. Such *distributed* systems are generally composed of several processes, running in parallel on one or several platforms, communicating with one

another to perform the required control task. Several communication schemes can be used. If the processes run on the same machine, they can communicate by manipulating shared variables. Processes running on different machines, on the other hand, can communicate asynchronously by sending messages to each other over a network.

Intrinsically, *distributed reactive control systems* are hard to design. Their distributed nature and/or the communication scheme they use can introduce subtle unforeseen behaviours. As argued above, when dealing with critical applications, such as plane control systems, or traffic light control systems, those unintended behaviours can have disastrous consequences. It is therefore essential, for the designer, to ensure the correctness of those systems. The first step towards that goal is to use an appropriate *development environment*, that will ease their development as much as possible.

Development Environment

One of the main challenges in the development of distributed applications is the design of the communication procedures. A development environment that takes care of that aspect is therefore most beneficial, because it allows the programmer to concentrate on the functional aspects of systems.

Classical solutions based on this idea exist, e.g. CORBA, DCOM, EJB [Tanenbaum and van Steen, 2002; Monson-Haefel, 2001]. Unfortunately, these solutions are quite heavy and completely hide all of the communication process, making the monitoring of such systems extremely difficult. More dedicated solutions with transparent distribution mechanisms, have been proposed. Examples of such solutions are distributed shared memory [Nitzberg and Lo, 1991] or, more specifically in the domain of control systems, synchronous languages like Esterel, Lustre or Signal [Aubry, 1997; Girault, 1994]. Unfortunately, even if shared memory solutions are generally lighter than the distributed objects ones, due to the cache coherence protocol, the time to access the memory can vary greatly and is not predictable.

This lead us to dSL (**d**istributed **S**upervision **L**anguage), a new environment and language designed to program distributed industrial control systems, providing transparent code distribution, and using low level mechanisms adapted for the industrial world. dSL, developed at *Macq Electronique*[1] in collaboration with the *Université Libre de Bruxelles* [De Wachter et al., 2003a; De Wachter et al., 2003b; De Wachter et al., 2005; De Wachter, 2005], offers advantages both to allow transparent distribution and, by the simplicity of the distribution mechanisms, to easily monitor the behaviours of

[1] a Belgian company specialized in industrial process control `http://www.macqel.be`

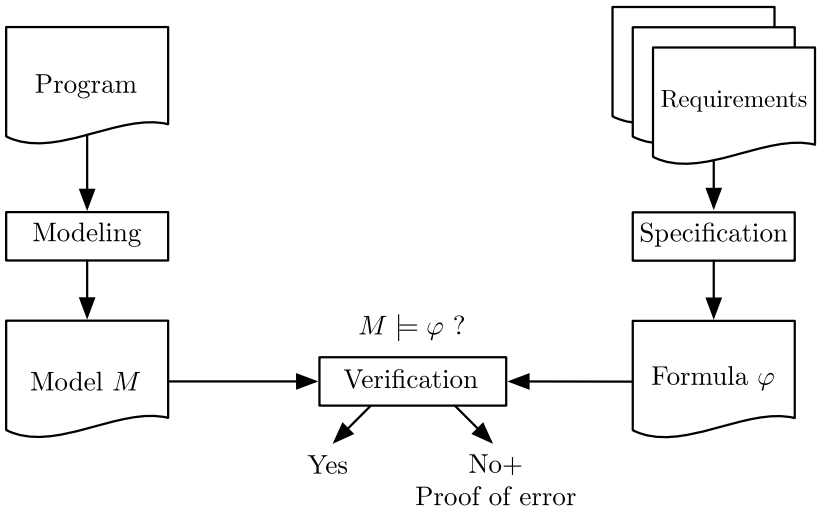


Figure 1 - The model checking process

the synthesized distributed system. This approach relieves the designer from a considerable burden, thus making the development of distributed reactive control systems less error-prone. However, this is not the panacea. Thorough validation is still needed. One of the most prominent techniques for that purpose is *model checking*.

Model Checking

Model Checking is an automatic technique for systematically verifying concurrent systems. It was first introduced in the early 1980s by Clarke and Emerson [Clarke and Emerson, 1981] and also independently by Queille and Sifakis [Queille and Sifakis, 1982]. As illustrated in Figure 1, the model checking process can be decomposed in three steps.

The first step is *modeling*, during which a formal, i.e. mathematical, model is extracted from the system. This model is generally some *finite-state* transition system like *Kripke structures* [Kripke, 1963] or finite automata. A lot of research efforts has also been put, in the past two decades, in the study of *infinite-state* models like *communicating finite state machines* [von Bochmann, 1978], *timed automata* [Alur and Dill, 1990], *well-structured transition systems* [Abdulla et al., 1996], *Petri nets* [Petri, 1962], *pushdown automata* [Bouajjani et al., 1997], etc.

The next step is *specification* and consists in stating the properties that the system must satisfy in order to be correct. Those properties are generally specified in some

sort of *temporal logic* [Pnueli, 1977; Sistla and Clarke, 1985]. There are mainly two categories of properties. The first is that of *safety* properties. Those are mostly *reachability* question, e.g "will something bad ever happen?". The other category is that of *liveness* properties, which are mostly responsiveness questions, e.g. "will something good eventually happen?". The specification is, to be sure, a crucial step of the model checking process. Indeed, model checking provides ways to determine if a system satisfies a given specification. However, there is no guarantees that the given specification is *correct*, i.e. that the specification describes an actual problem, or *complete*, i.e. that all potential problems are covered. One must therefore pay particular attention to designing good specifications. The last step is the actual *verification* where the model is checked against the specification. This results in a *yes/no* answer indicating if the model satisfies the given specification. Moreover, if the answer is negative, the user is usually provided with a proof that the system does not satisfy the specification. This proof, often given as an error trace, can then be used to identify the source of the problem. Ideally, this step is completely automated. Various model checking tools have been developed for that purpose, such as Spin [Holzmann, 2004], SMV [McMillan, 1993], or Uppaal [Bengtsson et al., 1995]. However, in practice, this verification often involves human assistance. For instance, an error trace provided by the model checker is generally extensive and quite complicated. It can also be *spurious*, i.e. the result of an error in the *model*. In any case, a meticulous analysis of this error trace is needed in order to find the cause of the problem. Model checking is an elegant approach to validation, and has several advantages. However, modeling complex distributed programs as transition systems has an inherent disadvantage known as the *state explosion problem*. The problem is that the number of states of such models grows exponentially with the number of components of the systems. When dealing with large distributed systems, this can prevent the designer from an exhaustive verification, even with efficient exploration techniques such as *partial order reduction* [Godefroid, 1996; Valmari, 1993] or *symbolic* model checking [McMillan, 1993]. Thankfully, when this happens, all hope is not lost. The designer can turn back to *testing*.

Testing

Testing was first regarded[2] as a separate task by Glenford J. Myers [Myers, 1979]. Until then, almost no distinction was made between the debugging of a program, i.e. making sure that the program works as the programmer intended, and the testing of a

[2]according to `http://en.wikipedia.org/wiki/Software_testing`

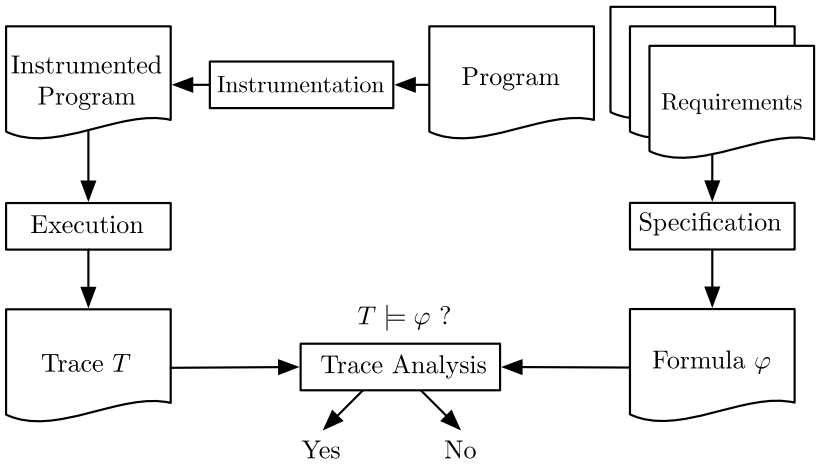


Figure 2 - The testing process

program, i.e. making sure that program works correctly. Since then, testing has come a long way, and is now widely accepted in the industrial world. Probably, one of the main reasons for that is that it scales very well to large systems.

Depending on the assumptions the developer makes on the system, testing can be categorized as either *black-box* or *white-box*. Black-box testing assumes an external view of the system, namely its inputs and outputs, whereas white-box testing assumes a substantial knowledge about the program under test, e.g. access to its source code. Several *levels* of testing can also be considered. At the component level, *unit* testing consists in inspecting each module of the software individually to determine if it has been implemented properly. At the intermediate level, *integration* testing checks for the presence of bugs introduced by the way modules interact with each other. At the global level, *system* testing examines the system under test as a whole in order to determine if it meets its requirements. Finally, at the user level, *acceptance* testing is conducted by the end-user, or client, to decide if the software is accepted or not.

In this work, we will concentrate our attention on checking that the system meets its requirements, i.e. *system* testing. We will also assume access to the internals of the system under test, i.e. *white-box* testing. In this particular context, as illustrated in Figure 2, the testing process can be decomposed in several steps.

The first two steps are *instrumentation* and *execution*. The system under test is instrumented to emit certain events. Of course the relevance of those events depends on the specifications. Then, the system is executed, possibly in a controlled environment and the events emitted by the system are collected together to form an execution

trace. The next step, as in model checking, is *specification*, where the requirements are formalized. In this study, we will use the same formalism as for model checking, namely formal logic. This is, however, not always the case. Other formalisms like UML use-case or sequence diagrams [Oestereich, 2002] can be used. Finally, the last step is *trace analysis* where the trace is examined to determine if it meets the specification. This step can be done automatically or manually, in which case, the specification step can be overlooked.

Contrarily to model-checking, testing is not an exhaustive technique. This implies that some bugs might be left unnoticed. However, if performed on a large number of traces, it can still give a reasonable confidence in the correctness of the system. Moreover, software testing can be used to unveil certain implementation errors that were not previously detected using model checking. Indeed, as mentioned earlier, model checking works on a *model* of the system. If the model is too coarse, it might therefore abstract away some errors, which would pass undetected.

Structure of this Thesis

The remainder of this dissertation is structured as follows.

Chapter 1 - Preliminaries. In the first chapter, we briefly present the few fundamental mathematical concepts and results that will be needed throughout this study.

Chapter 2 - Distributed Supervision Language. In the next chapter, we introduce dSL (**d**istributed **S**upervision **L**anguage), the development environment and programming language dedicated for distributed reactive control systems. First, we present the various constructs of the language. Then, we briefly detail one of its main feature: the automatic distribution. Finally, we discuss some related works. The content of this chapter is based on a joint work with Bram De Wachter and Thierry Massart and published in the following two articles.

[De Wachter et al., 2003a] De Wachter, B., Massart, T., and Meuter, C. (2003a). An Experiment on Synthesis and Verification of an Industrial Process Control in the dSL Environment. In *Proceeding of the 3rdInternational Workshop on Automated Verification of Critical Systems (AVoCS'03), Southampton (UK).*

[De Wachter et al., 2003b] De Wachter, B., Massart, T., and Meuter, C. (2003b). dSL: An Environment with Automatic Code Distribution for Industrial Control Systems.

In *Proceedings of the 7thInternational Conference on Principles of Distributed Systems (OPODIS'03), La Martinique (France)*, volume 3144 of *Lecture Notes in Computer Sciences*, pages 132–145. Springer.

Chapter 3 - Model Checking dSL Programs. Then, we study the problem of model checking dSL programs. For that purpose, we start by introducing a formal operational semantics for dSL programs. We then study the properties of this model, and show their impact in the context of model-checking. Finally, we explain how the model-checking can be performed in practice. Most of this chapter is based on a joint work with Bram De Wachter, Alexandre Genon and Thierry Massart, published in the following article.

[De Wachter et al., 2005] De Wachter, B., Genon, A., Massart, T., and Meuter, C. (2005). The Formal Design of Distributed Controllers with dSL and Spin. *Formal Aspect of Computing*, 17(2):177–200.

Chapter 4 - Testing dSL Programs In this chapter, we examine how the test of distributed concurrent systems in general, and dSL programs in particular, can be achieved. For that purpose, we explain how a dSL program can be instrumented to collect a distributed execution trace. Then, in order to capture these distributed execution traces, we introduce the model of partial order trace. Furthermore, we examine how to specify temporal and non-temporal properties on this model. This leads to define the problem of determining whether a given partial order trace satisfies a given property. In the case of non-temporal properties, this problem is known as the predicate detection problem, for which we give the theoretical complexity. In the case of temporal properties, we call this problem the trace checking problem, for which we also give the theoretical complexity. The content of this chapter is based on a joint work with Thierry Massart and Laurent Van Begin, which has been accepted for publication.

[Massart et al., 2007] Massart, T., Van Begin, L., and Meuter, C. (2007). On the Complexity of Partial Order Trace Model Checking. Accepted for publication in *Information Processing Letters*, to appear

Chapter 5 - Predicate Detection Then, we review the work that has been accomplished on the predicate detection problem and adapt it to our framework. In particular, we identify several classes of predicates for which the predicate detection problem can be solved efficiently.

Chapter 6 - LTL Trace Checking In the next chapter, we study how the trace checking problem for the LTL formulae can be solved. For that purpose, we first adapt traditional automata-based algorithms used for model checking. Then, we build on this solution, in order to come up with a symbolic solution that is much more efficient in practice. The content of this chapter in based on a joint work with Alexandre Genon and Thierry Massart, published in the following article.

[Genon et al., 2006] Genon, A., Massart, T., and Meuter, C. (2006). Monitoring Distributed Controllers: When an Efficient LTL Algorithm on Sequences Is Needed to Model-Check Traces. In *Proceedings of the 14thInternational Symposium on Formal Methods (FM'06), Hamilton (Canada)*, volume 4085 of *Lecture Notes in Computer Science*, pages 557–572. Springer-Verlag.

Chapter 7 - CTL Trace Checking In the following chapter, we study how the trace checking problem for the CTL formulae can be solved. We first examine a particular class of formulae that have already been studied in the literature, and review the existing algorithms. Then, we explain how to extend this work in order to account for all CTL formulae. The content of this chapter is based on a joint work with Gabriel Kalyon, Thierry Massart and Laurent Van Begin, published in the following article.

[Kalyon et al., 2007] Kalyon, G., Massart, T., Meuter, C., and Van Begin, L. (2007). Testing Distributed Systems Through Symbolic Model Checking. In *Proceeding of the 27thIFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'07), Tallinn (Estonia)*, volume 4574 of *Lecture Notes in Computer Science*, pages 263–279. Springer-Verlag.

Chapter 8 - Case Study: A Canal Lock Controller In the last chapter of this dissertation, we illustrate how the methods described in the previous chapters are used in practice on a concrete case study. More precisely, we study the design of a canal locks controller and its validation using model checking and testing.

Finally, we conclude this dissertation by summarizing our work, emphasizing our personal contributions, and presenting directions for future works.

Chapter

1

Preliminaries

« *The worst programs are the ones where the programmers doing the original work don't lay a solid foundation.* »

Bill Gates

In this chapter, we lay the groundwork for our work, by recalling the necessary mathematical concepts that will be used in the remainder of this dissertation. First, after recalling some basic notions on sets, relations and functions, in Section 1.1, we present a short introduction to lattice theory, in Section 1.2. Next, in Section 1.3, we focus on Boolean logics. We follow, in Section 1.4, with a few words on automata and formal languages. Finally, in Section 1.5, we present a brief introduction to model checking. A significant part of this chapter is taken from [Schneider, 2004].

1.1 Sets, Relations and Functions

The set of *natural* numbers $\{0, 1, 2, \ldots\}$ is noted $\mathbb{N}$. The set of *integer* numbers $\{0, 1, 2, \ldots\} \cup \{-1, -2, \ldots\}$ is noted $\mathbb{Z}$. We also note $\mathbb{N}^\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{+\infty\}$ and $\mathbb{Z}^\infty \stackrel{\text{def}}{=} \mathbb{Z} \cup \{-\infty, +\infty\}$. The natural order on numbers is extended on $\mathbb{N}^\infty$ and $\mathbb{Z}^\infty$ as expected. Given two integer numbers $a, b \in \mathbb{Z}^\infty$, the *interval* between a and b, is defined as $[a, b] \stackrel{\text{def}}{=} \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. We note $\mathbb{I}$ the set of such intervals. Moreover, given an interval $[a, b] \in \mathbb{I}$, we note $(a, b] \stackrel{\text{def}}{=} [a, b] \setminus \{a\}$, and $[a, b) \stackrel{\text{def}}{=} [a, b] \setminus \{b\}$. Finally, the set of Boolean values $\{\mathsf{tt}, \mathsf{ff}\}$, where tt, respectively ff, denotes *true*, respectively *false*, is noted $\mathbb{B}$. For a *Boolean* value $b \in \mathbb{B}$, we note $\overline{b}$ its *complement*, i.e. $\overline{b} \stackrel{\text{def}}{=} \mathsf{ff}$ iff $b = \mathsf{tt}$.

We note $\varnothing$ the empty set. Given a set X, we note 2^X the *powerset* of X, i.e. set of all subsets of X and X^k the set of k-uples of elements of X. For a k-uple $t \in X^k$, we note $t[i]$ its i^{th} component. Given two k-uples of integers $t, t' \in \mathbb{Z}^k$, we note $t \leq t'$ if and only if $\forall i \in [1,k] : t[i] \leq t'[i]$. The notion of interval is extended to k-uples as follows. Given two k-uples $a, b \in (\mathbb{Z}^\infty)^k$ such that $a \leq b$, the interval between a and b is defined as $[a,b] \stackrel{\text{def}}{=} \{t \in \mathbb{Z}^k \mid a \leq t \leq b\}$. The set of such intervals in k dimensions, also called *multi-rectangles*, is noted $\mathbb{M}(k)$.

Let us also recall the notion of *partition*. Given a set X, a partition P of X is a set of non-empty subsets of X such that $\forall Y, Z \in P : Y \cap Z = \varnothing$ and $\bigcup_{Y \in P} Y = X$. We note $\Pi(X)$ the set of partitions of X. Given, two partitions $P, P' \in \Pi(X)$, we say that P is *coarser* than P', or equivalently that P' is *finer* than P, noted $P \preceq P'$ if and only if $\forall X' \in P', \exists X \in P : X' \subseteq X$.

Next, let us recall a few concepts on relations. A binary relation $R \subseteq X \times X$ is *reflexive* if and only if $\forall x \in X : \langle x, x \rangle \in R$. Moreover, R is *antisymmetric* if and only if $\forall x, y \in X : (\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R) \Rightarrow (x = y)$. Furthermore, R is *transitive* if and only if $\forall x, y, z \in X : (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R) \Rightarrow (\langle x, z \rangle \in R)$. Finally, R is *total* if and only if $\forall x, y \in X : (\langle x, y \rangle \in R) \vee (\langle y, x \rangle \in R)$. We also sometimes use the infix notation for binary relations where $\langle x, y \rangle \in R$ is noted $x \, R \, y$.

Given a set X and a binary relation $R \subseteq X \times X$, we note $R^{-1} \stackrel{\text{def}}{=} \{\langle x, y \rangle \in X \times X \mid \langle y, x \rangle \in R\}$, its inverse. We also note $R^0 \stackrel{\text{def}}{=} \{\langle x, x \rangle \in X \times X \mid x \in X\}$ and for any $i \in \mathbb{N}$, $R^{i+1} \stackrel{\text{def}}{=} \{\langle x, y \rangle \in X \times X \mid \exists z \in X : \langle x, z \rangle \in R^i \wedge \langle z, y \rangle \in R\}$. The *transitive closure* of R is defined as $R^+ \stackrel{\text{def}}{=} \bigcup_{i \in [1, +\infty)} R^i$ and the *reflexive and transitive closure* is defined as $R^* = R^+ \cup R^0$.

We now turn our attention to functions. Given two sets X and Y, we note $X \mapsto Y$, the set of all functions whose *domain* is X and whose *co-domain* is Y. Given a function $f \in X \mapsto Y$, a subset $Z \subseteq X$ and an element $y \in Y$, we define $f[Z \mapsto y] \in X \mapsto Y$ as follows:

$$f[Z \mapsto y](z) \stackrel{\text{def}}{=} \begin{cases} y & \text{if } z \in Z \\ f(z) & \text{otherwise} \end{cases}$$

If Z is a singleton $\{z\}$, we shortcut $f[\{z\} \mapsto y]$ by $f[z \mapsto y]$. Furthermore, given two functions $f \in X \mapsto Y$ and $g \in Z \mapsto Y$ such that $X \cap Z = \varnothing$, we define $(f \cup g) \in (X \cup Z) \mapsto Y$ as follows:

$$(f \cup g)(w) \stackrel{\text{def}}{=} \begin{cases} f(w) & \text{if } w \in X \\ g(w) & \text{if } w \in Z \end{cases}$$

We also sometimes use Church's notation [Church, 1936] where a function $f \in X \mapsto Y$

is noted $\lambda x \cdot f(x)$. Given a function $f \in X \mapsto X$, we note $f^0 \stackrel{\text{def}}{=} \lambda x \cdot x$ and for any $i \in \mathbb{N}$, $f^{i+1} \stackrel{\text{def}}{=} \lambda x \cdot f(f^i(x))$.

A function $f \in X \mapsto Y$ is *injective* if and only if $\forall x, x' \in X : (f(x) = f(x')) \Rightarrow (x = x')$. Moreover, f is *surjective* if and only if $\forall y \in Y, \exists x \in X : f(x) = y$. Finally, f is *bijective* if and only if it is both injective and surjective. Given two sets X, Y, and two binary relation $R \subseteq X \times X$ and $S \subseteq Y \times Y$, a bijective function $f \in X \mapsto Y$ is an *isomorphism* of $\langle X, R\rangle$ onto $\langle Y, S\rangle$ if and only if $\forall x, x' \in X : (x\, R\, x') \Leftrightarrow (f(x)\, S\, f(x'))$.

1.2 Lattice Theory

In this section, we give a brief introduction to lattice theory. First, in Section 1.2.1, we introduce partially ordered sets. Next, we turn our attention to lattices in Section 1.2.2. Finally, we conclude in Section 1.2.3, with a few words on fixed points computation.

1.2.1 Partially Ordered Set

A *partially ordered set*, or poset for short, is a tuple $\langle X, \sqsubseteq\rangle$ where X is a set of elements and $\sqsubseteq \subseteq X \times X$ is a partial order, i.e. a binary, reflexive, antisymmetric and transitive relation on X. Given a poset $\langle X, \sqsubseteq\rangle$, and a subset $Y \subseteq X$, we will often abusively note $\langle Y, \sqsubseteq\rangle$ as a shortcut for $\langle Y, \sqsubseteq \cap (Y \times Y)\rangle$. A poset $\langle X, \sqsubseteq\rangle$, is a *totally ordered set* if $\sqsubseteq$ is total.

Given poset $\langle X, \sqsubseteq\rangle$, an element $x \in X$ is *minimal* in X if and only if $\forall x' \in X : (x' \sqsubseteq x) \Rightarrow (x = x')$. Dually, an element $x \in X$ is *maximal* in X if and only if $\forall x' \in X : (x \sqsubseteq x') \Rightarrow (x = x')$. We note $\mathsf{Min}_{\sqsubseteq}(X)$, respectively $\mathsf{Max}_{\sqsubseteq}(X)$, the set of minimal, respectively maximal, elements of X. An element $x \in X$ is the *least*, respectively *greatest*, element of X, if $\mathsf{Min}_{\sqsubseteq}(X) = \{x\}$, respectively if $\mathsf{Max}_{\sqsubseteq}(X) = \{x\}$. We note $\mathsf{min}_{\sqsubseteq}(X)$, respectively $\mathsf{max}_{\sqsubseteq}(X)$, the least, respectively greatest element of X, if it exists.

Given a set $Y \subseteq X$, an element $z \in X$ is an *upper bound* of Y if and only if $\forall y \in Y : y \sqsubseteq z$. We note $\mathsf{UB}_{\sqsubseteq}(Y)$ the set of upper bounds of Y. Similarly, an element $z \in X$ is a *lower bound* of Y if and only if $\forall y \in Y : z \sqsubseteq y$. We note $\mathsf{LB}_{\sqsubseteq}(Y)$ the set of lower bounds of Y. The *least upper bound*, respectively *greatest lower bound*, if it exists, is defined as $\mathsf{lub}_{\sqsubseteq}(Y) \stackrel{\text{def}}{=} \mathsf{min}_{\sqsubseteq}(\mathsf{UB}_{\sqsubseteq}(Y))$, respectively $\mathsf{glb}_{\sqsubseteq}(Y) \stackrel{\text{def}}{=} \mathsf{max}_{\sqsubseteq}(\mathsf{LB}_{\sqsubseteq}(Y))$.

A subset $Y \subseteq X$ is a *chain* if and only if Y contains only comparable elements, i.e. $\forall y, y' \in Y : (y \sqsubseteq y') \vee (y' \sqsubseteq y)$. Conversely, Y is an *antichain* of X, if and only if Y contains only incomparable elements, i.e. $\forall x, y \in Y : (x \neq y) \Rightarrow (x \not\sqsubseteq y \wedge y \not\sqsubseteq x)$.

The width of a poset, noted $\mathsf{width}_{\sqsubseteq}(X)$, is the size of a largest antichain of X, i.e. an antichain of maximal size.

Furthermore, given an element $x \in X$, we define the *downward closure* of x as $\downarrow x \stackrel{\text{def}}{=} \{x' \in X \mid x' \sqsubseteq x\}$. Symmetrically, we define the *upward closure* of x as $\uparrow x \stackrel{\text{def}}{=} \{x' \in X \mid x \sqsubseteq x'\}$. The downward and upward closures are extended to sets as expected. Given a subset $Y \subseteq X$, $\downarrow Y \stackrel{\text{def}}{=} \bigcup_{y \in Y} \downarrow y$ and $\uparrow Y \stackrel{\text{def}}{=} \bigcup_{y \in Y} \uparrow y$. A subset $Y \subseteq X$ is *downward closed*, respectively *upward closed*, if and only if $\downarrow Y = Y$, respectively $\uparrow Y = Y$. Let $\mathsf{DC}_{\sqsubseteq}(X) \subseteq 2^X$, respectively $\mathsf{UC}_{\sqsubseteq}(X) \subseteq 2^X$, denote the set of downward, respectively upward, closed subsets of X. Downward closed subset of X are also known as *order ideals* of X.

Given two posets $\langle X, \sqsubseteq_X \rangle$ and $\langle Y, \sqsubseteq_Y \rangle$, a function $f \in X \mapsto Y$ is *monotonic* if and only if $\forall x, x' \in X : (x \sqsubseteq_X x') \Rightarrow (f(x') \sqsubseteq_Y f(y'))$. The function f is *continuous* if and only if it is monotonic and if for every non-empty chain $Z \subseteq X$, $f(\mathsf{lub}_{\sqsubseteq_X}(Z)) = \mathsf{lub}_{\sqsubseteq_Y}(\{f(x) \mid x \in Z\})$ and $f(\mathsf{glb}_{\sqsubseteq_X}(Z)) = \mathsf{glb}_{\sqsubseteq_Y}(\{f(x) \mid x \in Z\})$. Note that for finite posets, any monotonic function is also continuous.

Example 1.1

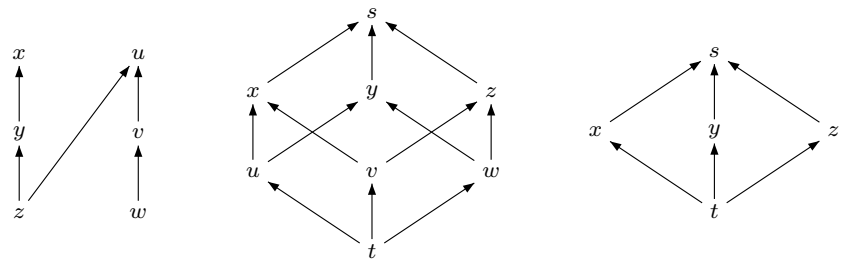
The function $f \in \mathbb{N} \mapsto \mathbb{N}$ defined as $f(x) \stackrel{\text{def}}{=} x+1$ for any $x \in \mathbb{N}$, is both monotonic and continuous in the poset $\langle \mathbb{N}, \leq \rangle$. On the other hand, assuming a set $X \stackrel{\text{def}}{=} \{\frac{n}{n+1} \mid n \in \mathbb{N}\} \cup \{1\}$, the function $g \in X \mapsto X$ defined as

$$g(x) \stackrel{\text{def}}{=} \begin{cases} \frac{1}{2} & \text{if } x \neq 1 \\ 1 & \text{otherwise} \end{cases}$$

is monotonic in $\langle X, \leq \rangle$, but not continuous. Indeed, consider the non-empty chain $Z = X \setminus \{1\}$. We have that $g(\mathsf{lub}_{\leq}(Z)) = g(1) = 1$, while $\mathsf{lub}_{\leq}(\{g(x) \mid x \in Z\}) = \mathsf{lub}_{\leq}(\{\frac{1}{2}\}) = \frac{1}{2}$.

1.2.2 Lattice

A poset $\langle X, \sqsubseteq \rangle$ is a *lattice* if and only if $\forall x, y \in X$, $\mathsf{lub}_{\sqsubseteq}(\{x, y\})$ and $\mathsf{glb}_{\sqsubseteq}(\{x, y\})$ exist. Moreover, $\langle X, \sqsubseteq \rangle$ is a *complete lattice* if and only if for every non-empty subset $Y \subseteq X$, $\mathsf{lub}_{\sqsubseteq}(Y)$ and $\mathsf{glb}_{\sqsubseteq}(Y)$ exists. In particular, a complete lattice $\langle X, \sqsubseteq \rangle$ admits a greatest element $\mathsf{max}_{\sqsubseteq}(X) = \mathsf{lub}_{\sqsubseteq}(X)$ and a least element $\mathsf{min}_{\sqsubseteq}(X) = \mathsf{glb}_{\sqsubseteq}(X)$. Note that any finite lattice is trivially complete. Given a lattice $\langle X, \sqsubseteq \rangle$, the *join*, respectively *meet*, of two elements $x, y \in X$, noted $x \sqcup y$, respectively $x \sqcap y$, is defined as $\mathsf{lub}_{\sqsubseteq}(\{x, y\})$, respectively $\mathsf{glb}_{\sqsubseteq}(\{x, y\})$. A lattice $\langle X, \sqsubseteq \rangle$ is *distributive* if and only if its meet operator distributes over its join operator, i.e. $\forall x, y, z \in X : x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (y \sqcup z)$.



(a) (b) (c)

Figure 1.1 - Examples of posets

Finite posets, and therefore finite lattices, are often represented graphically using *Hasse diagrams*. The Hasse diagram of a poset $\langle X, \sqsubseteq \rangle$ is a directed acyclic graph $\langle X, R \rangle$ such that $\langle x, y \rangle \in R$ if and only if x and y are strictly ordered, i.e. $x \sqsubseteq y \wedge x \neq y$ and if they are direct successors in the order, i.e. $\nexists z \in X \setminus \{x, y\} : x \sqsubseteq z \sqsubseteq y$. Note that if $\langle X, R \rangle$ is the Hasse diagram of a poset $\langle X, \sqsubseteq \rangle$, we have that $\sqsubseteq = R^*$.

Example 1.2

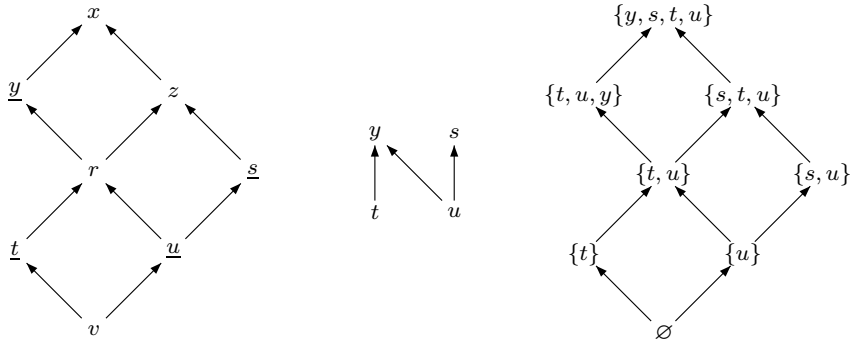
Figure 1.1 shows the Hasse diagram of three posets. The first poset, in Figure 1.1(a), is not a lattice. Indeed, consider e.g. elements y and u. Their least upper bound, namely $y \sqcup u$, does not exist. On the other hand, the two other posets, presented in Figure 1.1(b) and Figure 1.1(c), are lattices. The first of those two, in Figure 1.1(b), is distributive, whereas, the other one, in Figure 1.1(c) is not, since $x \sqcap (y \sqcup z) = x \sqcap s = x \neq (x \sqcap y) \sqcup (x \sqcap z) = t \sqcup t = t$.

Given a poset $\langle X, \sqsubseteq \rangle$, it is well known in lattice theory that the set of order ideals of X along with set inclusion forms a complete distributive lattice.

Theorem 1.1 (Lattice of Order Ideals [Priestley and Davey, 2002])

Given a poset $\langle X, \sqsubseteq \rangle$, we have that $\langle \mathsf{DC}_{\sqsubseteq}(X), \subseteq \rangle$ forms a complete distributive lattice.

Another well known result in lattice theory is that there exists a duality between finite posets and distributive lattices. This result is known as Birkhoff's representation theorem. This theorem is based on the notion of *join-irreducible* elements. Formally, given a lattice $\langle X, \sqsubseteq \rangle$, an element $x \in X$ is *join-irreducible*, if and only if (i) it is not the least element, i.e. $\exists y \in X : (y \neq x) \wedge (y \sqsubseteq x)$, and (ii) it cannot be expressed as the join of two elements both different from itself, i.e. $\forall y, y' \in X : (x = y \sqcup y') \Rightarrow (x = y \vee x = y')$. We note $\mathsf{JI}_{\sqsubseteq}(X)$ the set of join-irreducible elements of X. Birkhoff's



(a) $\langle X, \sqsubseteq \rangle$ (b) $\langle \mathsf{JI}_{\sqsubseteq}(X), \sqsubseteq \rangle$ (c) $\langle \mathsf{DC}_{\sqsubseteq}(\mathsf{JI}_{\sqsubseteq}(X)), \subseteq \rangle$

Figure 1.2 - Illustration of Birkhoff's representation theorem

representation theorem can then be formulated as follows.

Theorem 1.2 (Representation of Finite Distributive Lattice [Brikhoff, 1940])

Let $\langle X, \sqsubseteq \rangle$ be a finite distributive lattice. The function $f \in X \mapsto \mathsf{DC}_{\sqsubseteq}(\mathsf{JI}_{\sqsubseteq}(X))$ defined as $f \stackrel{\text{def}}{=} \lambda x \cdot \downarrow x \cap \mathsf{JI}_{\sqsubseteq}(X)$ is an isomorphism of $\langle X, \sqsubseteq \rangle$ onto $\langle \mathsf{DC}_{\sqsubseteq}(\mathsf{JI}_{\sqsubseteq}(X)), \subseteq \rangle$. Dually, let $\langle Y, \sqsubseteq \rangle$ be a finite poset. The function $g \in Y \mapsto \mathsf{JI}_{\sqsubseteq}(\mathsf{DC}_{\sqsubseteq}(Y))$ defined as $g \stackrel{\text{def}}{=} \lambda y \cdot \downarrow y$ is an isomorphism of $\langle Y, \sqsubseteq \rangle$ onto $\langle \mathsf{JI}(\mathsf{DC}(Y)), \subseteq \rangle$.

Intuitively, Theorem 1.2 states that we can go from a finite distributive lattice to a finite poset by keeping only join irreducible elements, and conversely, we can go from a finite poset to a finite distributive lattice by constructing the set of order ideals.

Example 1.3

Figure 1.2 illustrates the duality between posets and distributive lattices. First, Figure 1.2(a) shows a distributive lattice $\langle X, \sqsubseteq \rangle$. For clarity, the join-irreducible elements have been underlined. Then, Figure 1.2(b) shows the poset constructed by keeping only the join irreducible elements. Finally, Figure 1.2(c) shows the lattice of order ideals built on this poset. This lattice is clearly isomorphic to the one from Figure 1.2(a).

1.2.3 Fixed Points

Given a function $f \in X \mapsto X$, an element $x \in X$ is a *fixed point* of f if and only if $x = f(x)$. We note $\mathsf{FP}(f) \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$ the set of fixed points of f. We now state some well-known results on fixed points computation.

```
1  function computeLFP(X, ⊑, f)
   input  : a complete lattice ⟨X, ⊑⟩, a continuous function f ∈ X ↦ X
   returns: lfp_⊑(X)
2  begin
3      x := glb_⊑(X)
4      repeat
5          x' := x, x := f(x)
6      until x = x'
7      return x
8  end

9  function computeGFP(X, ⊑, f)
   input  : a complete lattice ⟨X, ⊑⟩, a continuous function f ∈ X ↦ X
   returns: gfp_⊑(X)
10 begin
11     x := lub_⊑(X)
12     repeat
13         x' := x, x := f(x)
14     until x = x'
15     return x
16 end
```

Algorithm 1.1 - Fixed point computation in a complete lattice

Theorem 1.3 (Fixed Point Characterisation [Tarski, 1955; Knaster, 1928])
Given a complete lattice $\langle X, \sqsubseteq \rangle$ and a continuous function $f \in X \mapsto X$, we have that $\langle \mathsf{FP}(f), \sqsubseteq \rangle$ is a complete lattice. In particular, the *least fixed point*, noted $\mathsf{lfp}_{\sqsubseteq}(f)$, and the *greatest fixed point*, noted $\mathsf{gfp}_{\sqsubseteq}(f)$, can be characterized as follows:

$$\begin{array}{rcl} \mathsf{lfp}_{\sqsubseteq}(f) & \stackrel{\mathbf{def}}{=} & \mathsf{lub}_{\sqsubseteq}(\mathsf{FP}(f)) \;=\; \mathsf{lub}_{\sqsubseteq}(\{x \in X \mid f(x) \sqsubseteq x\}) \\ \mathsf{gfp}_{\sqsubseteq}(f) & \stackrel{\mathbf{def}}{=} & \mathsf{glb}_{\sqsubseteq}(\mathsf{FP}(f)) \;=\; \mathsf{glb}_{\sqsubseteq}(\{x \in X \mid x \sqsubseteq f(x)\}) \end{array}$$

Furthermore, given an element $x \in X$ s.t. $x \sqsubseteq f(x)$ and $x \sqsubseteq \mathsf{lfp}(f)$, the sequence $f^i(x)$, for $i \in \mathbb{N}$, converges to $\mathsf{lfp}_{\sqsubseteq}(f)$. Symmetrically, given an element $y \in X$ s.t. $f(y) \sqsubseteq y$ and $\mathsf{gfp}_{\sqsubseteq}(f) \sqsubseteq y$, the sequence $f^i(y)$, for $i \in \mathbb{N}$, converges to $\mathsf{gfp}_{\sqsubseteq}(f)$.

In particular, the sequence $f^i(\mathsf{glb}_{\sqsubseteq}(X))$, respectively $f^i(\mathsf{lub}_{\sqsubseteq}(X))$, for $i \in \mathbb{N}$ converges to the least, respectively greatest, fixed point of f. This yields natural semi-

algorithms for computing the least and greatest fixed points, as shown in Algorithm 1.1. Note that these are only semi-algorithms, i.e. they may not terminate. However, for *finite* complete lattices, they will.

1.3 Boolean Logics

In this section, we briefly recall the syntax and semantics of the propositional Boolean logic and the quantified Boolean logic. We also recall the complexity of the associated satisfiability problem.

1.3.1 Propositional Boolean Logic

Let us first recall the syntax of the propositional Boolean logic.

Definition 1.1 (Syntax of Propositional Boolean Logic)

Given a set of propositions $\mathbb{P}$, a formula in Propositional Boolean Logic (PBL) is defined using the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi$$

where $p \in \mathbb{P}$

In the previous definition, $\top$ is the *true* formula, $\neg$ is the *negation* operator, $\vee$ is the *disjunction* operator. Moreover, on top of this core syntax, other derived operators are defined as usual, such as *conjunction* $(\varphi \wedge \psi) \stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi)$, *implication* $(\varphi \Rightarrow \psi) \stackrel{\text{def}}{=} (\neg\varphi \vee \psi)$ and *equivalence* $(\varphi \Leftrightarrow \psi) \stackrel{\text{def}}{=} (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$. Moreover, one can define the *false* formula $\bot \stackrel{\text{def}}{=} \neg\top$. A *literal* is a formula of the form p or $\neg p$ for some proposition $p \in \mathbb{P}$. A PBL formula φ is in *disjunctive normal form* (DNF) if and only if φ is a disjunction of conjunctions of literals. Conversely, a PBL formula φ is in *conjunctive normal form* (CNF) if and only if φ is a conjunction of disjunctions of literals. Moreover, a formula φ is in *negation normal form* (NNF) if and only if negations only precede propositions in φ. Note that CNF, DNF and NNF are *normal forms*, i.e. to every PBL formula, corresponds exactly one formula in CNF, respectively DNF and NNF. In the following, given a PBL formula φ, we note $\mathsf{DNF}(\varphi)$, $\mathsf{CNF}(\varphi)$ and $\mathsf{NNF}(\varphi)$, the disjunctive, conjunctive and negation normal form of φ. The size of a PBL formula φ, noted $|\varphi|$, is defined as the number of propositions and operators used to build φ. We note $\mathsf{prop}(\varphi)$ the set of propositions appearing in φ. PBL formulae are interpreted over *valuations* of $\mathbb{P}$. For technical reasons, we prefer using subsets of $\mathbb{P}$ equivalently. The semantics is defined by the satisfaction relation defined hereafter.

Definition 1.2 (Semantics of PBL)

Given a set of propositions $\mathbb{P}$, let φ, ψ be two PBL formulae over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition and $V \subseteq \mathbb{P}$ be a subset of $\mathbb{P}$. The satisfaction relation for PBL, noted $\models_{\mathsf{P}}$, is defined inductively as follows:

$$\begin{array}{lll} V \models_{\mathsf{P}} \top & & \\ V \models_{\mathsf{P}} p & \text{iff} & p \in V \\ V \models_{\mathsf{P}} \neg\varphi & \text{iff} & V \not\models_{\mathsf{P}} \varphi \\ V \models_{\mathsf{P}} \varphi \vee \psi & \text{iff} & V \models_{\mathsf{P}} \varphi \text{ or } V \models_{\mathsf{P}} \psi \end{array}$$

A PBL formula φ is *satisfiable* if and only if there exists a subset $V \subseteq \mathbb{P}$ such that $V \models_{\mathsf{P}} \varphi$. Finally, given a PBL formula φ, the *satisfiability problem* for PBL (PBL-SAT) consists in determining if φ is satisfiable. The problem is known to be NP-complete.

Theorem 1.4 (Complexity of PBL-SAT [Cook, 1971])

The satisfiability problem for PBL is NP-complete.

For a good introduction to computational complexity, we refer the reader to [Papadimitriou, 1994] and (in french) [Wolper, 2001]. For a very complete list of complexity classes and their definitions, we refer the reader to [Aaronson and Kuperberg, 2005].

1.3.2 Quantified Boolean Logic

Let us next examine quantified Boolean logic. First, we recall the syntax.

Definition 1.3 (Syntax of Quantified Boolean Logic)

Given a set of propositions $\mathbb{P}$, a formula in Quantified Boolean Logic (QBL) is defined using the following grammar:

$$\varphi ::= \exists p\, \varphi \mid \forall p\, \varphi \mid \psi$$

where $p \in \mathbb{P}$ and ψ is a PBL formula over $\mathbb{P}$.

In the previous definition, $\exists$ is the *existential quantifier* and $\forall$ is the *universal quantifiers*. In the following, we will assume that each proposition is quantified at most once. A QBL formula is *fully quantified* if all propositions are quantified. Given a QBL formula $\varphi = Q_1 p_1\, Q_2 p_2 \ldots Q_r p_r\, \psi$, where $Q_i \in \{\forall, \exists\}$ and $p_i \in \mathbb{P}$ for any $i \in [1, r]$ and where ψ is a PBL formula, the size of φ is defined as $|\varphi| \stackrel{\text{def}}{=} |\psi|$. Similarly to PBL formulae, QBL formulae are also interpreted over subsets of $\mathbb{P}$. The semantics is defined by the satisfaction relation, extended from the propositional case.

Definition 1.4 (Semantics of QBL)

Given a set of propositions $\mathbb{P}$, let ψ be a QBL formula over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition and $V \subseteq \mathbb{P}$ be a subset of $\mathbb{P}$. The satisfaction relation for QBL, noted $\models_{\mathsf{Q}}$, is defined inductively as follows:

$$\begin{array}{lll} V \models_{\mathsf{Q}} \psi & \text{iff} & \psi \text{ is a PBL formula and } V \models_{\mathsf{P}} \psi \\ V \models_{\mathsf{Q}} \exists p\, \varphi & \text{iff} & V \models_{\mathsf{Q}} \varphi[p/\top] \text{ or } V \models_{\mathsf{Q}} \varphi[p/\bot] \\ V \models_{\mathsf{Q}} \forall p\, \varphi & \text{iff} & V \models_{\mathsf{Q}} \varphi[p/\top] \text{ and } V \models_{\mathsf{Q}} \varphi[p/\bot] \end{array}$$

where $\varphi[p/\chi]$ is the formula built from φ by replacing each instance of p by χ.

In this definition, for simplicity, we assume without loss of generality that all quantifiers appear at the beginning of the formula. Note that the truth value of a QBL formula only depends on the valuation of its *free*, i.e. not quantified, propositions. In particular, for fully quantified formulae, the satisfaction does not depend on V at all. Similarly to PBL, a QBL formula φ is satisfiable if and only if there exists a valuation $V \subseteq \mathbb{P}$ such that $V \models_{\mathsf{Q}} \varphi$. Given a QBL formula φ, the satisfiability problem for QBL (QBL-SAT) consists in deciding if φ is satisfiable. This problem is known to be PSPACE-complete, even for fully quantified QBL formulae.

Theorem 1.5 (Complexity of QBL-SAT [Stockmeyer and Meyer, 1973])

The satisfiability problem for QBL is PSPACE-complete.

1.4 Languages and Automata

An *alphabet* Σ is a set of symbol called *letters*. A *word* w over Σ is a finite or infinite sequence of letters of Σ. The empty word is noted ε. The set of finite words over Σ is noted Σ^*, and that of infinite words Σ^ω. We also note $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^* \setminus \{\varepsilon\}$ the set of non empty finite words. For a finite word $w = a_1 a_2 \dots a_n \in \Sigma^*$ the *length* of w, noted $|w|$, is defined as n. The length of an infinite word $w \in \Sigma^\omega$ is defined as $+\infty$. Given an infinite word $w \in \Sigma^\omega$, we note $\mathsf{inf}(w)$ the set of letters that appear infinitely often in w. It is sometimes convenient to view a word as a function $w \in [0, |w|) \mapsto \Sigma$, so that $w(0)$ is the first letter of w, $w(1)$ the second, and so on. For $x < |w|$, we define $w[x] \stackrel{\text{def}}{=} \lambda i \cdot w(i + x)$ the suffix of w starting at $w(x)$. A *language* over an alphabet Σ is a subset $L \subseteq \Sigma^* \cup \Sigma^\omega$. Given a finite word $w_1 \in \Sigma^*$ and a finite or infinite word $w_2 \in \Sigma^* \cup \Sigma^\omega$, we note their *concatenation* $w_1 \cdot w_2$. Given a subset $\Sigma' \subseteq \Sigma$, the *projection* of a word $w \in \Sigma^*$ on Σ', noted $w_{/\Sigma'}$, is obtained from w by keeping only symbols of Σ'. Formally, $w_{/\Sigma'}$ can be defined recursively as follows.

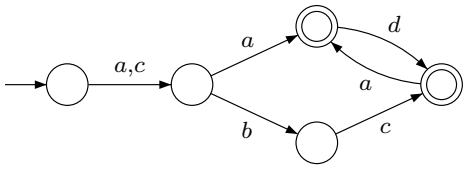


Figure 1.3 - Example of finite automata

$$w_{/\Sigma'} \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ w'_{/\Sigma'} & \text{if } (w = a \cdot w') \wedge (a \notin \Sigma') \\ a \cdot w'_{/\Sigma'} & \text{if } (w = a \cdot w') \wedge (a \in \Sigma') \end{cases}$$

The *shuffle product* of two finite words $w_1 \in \Sigma_1^*$ and $w_2 \in \Sigma_2^*$, noted $w_1 \parallel w_2$, is defined as $\{w \in (\Sigma_1 \cup \Sigma_2)^* \mid (w_{/\Sigma_1} = w_1) \wedge (w_{/\Sigma_2} = w_2)\}$.

Example 1.4

Given an alphabet $\Sigma = \{a, b, c, d\}$, we have that $abcbad_{/\{a,b\}} = abba$. Moreover, if $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{c, d, e\}$, we have that $acb \parallel ecd = \{aecbd, eacbd, aecdb, eacdb\}$. Finally note that if $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{b, c, d\}$, we have that $abc \parallel cbd = \varnothing$, because b and c do not appear in the same order in those two words.

Languages are often represented using finite automata, either on finite or infinite words. The formal definition follows.

Definition 1.5 (Finite Automaton)

A finite automaton, FA for short, is a tuple $A = \langle S, s_0, F, \Sigma, \Delta \rangle$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $F \subseteq S$ is the set of final states,
- Σ is a finite alphabet,
- $\Delta \subseteq S \times \Sigma \times S$ is the transition relation.

Given a FA $A = \langle S, s_0, F, \Sigma, \rightarrow \rangle$ and a state $s \in S$, we note $\mathsf{in}(s) \stackrel{\text{def}}{=} \{a \in \Sigma \mid \exists s' \in S : \langle s', a, s \rangle \in \Delta\}$ and $\mathsf{out}(s) \stackrel{\text{def}}{=} \{a \in \Sigma \mid \exists s' \in S : \langle s, a, s' \rangle \in \Delta\}$. Furthermore, given a symbol $a \in \Sigma$, we note $\mathsf{next}(s, a) \stackrel{\text{def}}{=} \{s' \in S \mid \langle s, a, s' \rangle \in \Delta\}$. A FA A is *deterministic* if and only if $\forall s \in S, \forall a \in \Sigma : |\mathsf{next}(s, a)| \leq 1$.

Example 1.5

Figure 1.3 shows a graphical representation of a finite automaton over the alphabet $\Sigma = \{a, b, c, d\}$. In this diagram, nodes represent states, and edges transitions between states. The initial state is the node marked with a small incoming arrow, and final states are the doubly circled nodes. Edges are labelled with one or more letter(s), each of which correspond to one transition. For instance, the outgoing edge from the initial state is labelled with a, c, indicating two transitions between this initial state s_0 and the next state s_1, i.e. $\{\langle s_0, a, s_1\rangle, \langle s_0, c, s_1\rangle\} \subseteq \Delta$.

A FA can be interpreted either over finite words, or over infinite words. In the former case, a finite word $w \in \Sigma^*$ is accepted by A if and only if there exists a finite sequence of states $\sigma \in S^*$ such that $|\sigma| = |w| + 1$, $\sigma(|w| + 1) \in F$ and $\forall i \in [0, |w|\,] : \langle \sigma(i), w(i), \sigma(i + 1)\rangle \in \Delta$. The set of finite words accepted by A is noted $\mathsf{lang}(A)$. In the latter case, an infinite word $w \in \Sigma^\omega$ is accepted by A if and only if there exists a infinite sequence of states $\sigma \in \Sigma^\omega$ such that $\mathsf{inf}(\sigma) \cap F \neq \varnothing$ and that $\forall i \in \mathbb{N} : \langle \sigma(i), w(i), \sigma(i + 1)\rangle \in \Delta$. The set of infinite words accepted by A is noted $\omega\text{-}\mathsf{lang}(A)$. This interpretation over infinite words is due to Büchi [Büchi, 1960].

1.5 Model Checking

As we already mentioned in the introduction, the model checking process can be decomposed in three steps. First, a mathematical model is extracted from the system by abstracting its behaviours. The resulting model is generally formalized in terms of a *Kripke structure*. This model is presented in Section 1.5.1. Then, the requirements on the system are specified, generally in some sort of *temporal logics*. Finally, in the verification step, the system is checked against the specification to determine whether or not the requirements are met. We examine how this can be done for two of the most commonly used temporal logics, namely LTL and CTL, in Section 1.5.2 and Section 1.5.3.

1.5.1 Kripke Structures

A Kripke structure [Kripke, 1963] is a mathematical model expressing how the truth value of Boolean propositions can evolve over time. This model is based on the notion of *transition systems*.

Definition 1.6 (Transition System)

A transition system, or TS for short, is a tuple $S = \langle Q, I, \rightarrow \rangle$ where:

- Q is a set of states,
- $I \subseteq Q$ is the set of initial states,
- $\rightarrow \subseteq Q \times Q$ is the transition relation.

If Q is finite, we say that S is *finite-state*. The *size* of a TS S is defined as $\mathsf{max}_{\leq}(\{|Q|, |\rightarrow|\})$ if S is finite-state, $+\infty$ otherwise. A *run* of S starting in a state $q \in Q$ is a finite or infinite word $\rho \in Q^+ \cup Q^\omega$ such that (i) $\rho(0) = q$ and (ii) $\forall i \in [0, |\rho| - 1) : \rho(i) \rightarrow \rho(i+1)$. A finite run ρ is *maximal* if and only if the last state of ρ has no outgoing transitions, i.e. $\nexists q \in Q : \rho(|\rho| - 1) \rightarrow q$. We note $\omega\text{-}\mathsf{runs}(q)$, respectively $\mathsf{runs}(q)$, the set of infinite, respectively finite maximal, runs starting in q. We also define $\omega\text{-}\mathsf{runs}(S) \stackrel{\text{def}}{=} \bigcup_{q \in I} \omega\text{-}\mathsf{runs}(q)$, respectively $\mathsf{runs}(S) = \bigcup_{q \in I} \mathsf{runs}(q)$, the set of infinite, respectively finite maximal, runs of S. We also note $\rightsquigarrow$ the reflexive and transitive closure of the transition relation, i.e. $\rightsquigarrow \stackrel{\text{def}}{=} (\rightarrow)^*$. Moreover, given a subset $X \subseteq Q$, we define $\mathsf{post}(X) \stackrel{\text{def}}{=} \{q' \in Q \mid \exists q \in X : q \rightarrow q'\}$ the set of direct successors of X, $\mathsf{pre}(X) \stackrel{\text{def}}{=} \{q' \in Q \mid \exists q \in X : q' \rightarrow q\}$, the set of direct predecessors of X and $\widetilde{\mathsf{pre}}(X) \stackrel{\text{def}}{=} \{q' \in Q \mid \forall q \in Q : (q' \rightarrow q) \Rightarrow q \in X\}$ the set of states where X cannot be avoided in one transition. Note that by definition, we have that $\forall X \subseteq Q : \widetilde{\mathsf{pre}}(X) = Q \setminus \mathsf{pre}(Q \setminus X)$. For singletons, we shortcut $\mathsf{post}(\{q\})$ by $\mathsf{post}(q)$, $\mathsf{pre}(\{q\})$ by $\mathsf{pre}(q)$ and $\widetilde{\mathsf{pre}}(\{q\})$ by $\widetilde{\mathsf{pre}}(q)$. We also note $\mathsf{reach}(S) = \{q \in Q \mid \exists q' \in I : q' \rightsquigarrow q\}$. We can now introduce *Kripke structures*, formalized as follows.

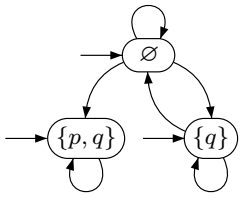
Definition 1.7 (Kripke Structure)

A Kripke structure, or KS for short, over a set of propositions $\mathbb{P}$ is a tuple $K = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ where $\langle Q, I, \rightarrow \rangle$ is a transition system and $\mathcal{L} \in Q \mapsto 2^{\mathbb{P}}$ is an interpretation function.

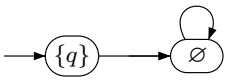
In other words, a Kripke struture is a transition systems where each state is labelled with the set of Boolean propositions true in this state.

Example 1.6

Figure 1.4 shows graphical representations of two Kripke structures. Similarly to finite automata, nodes represent states, edges transitions, and small incoming arrows mark initial states. Moreover, nodes are labelled with the result of the interpretation function.



(a) K_1 over $\{p, q\}$



(b) K_2 over $\{q\}$

Figure 1.4 - Examples of Kripke Structures

The *trace* of a run ρ, noted $\mathcal{L}(\rho)$, is a word σ over $2^{\mathbb{P}}$, also called a propositional sequence, such that $\forall i \in [0, |\rho|) : \sigma(i) = \mathcal{L}(\rho(i))$. We note respectively $\mathsf{traces}(K) \stackrel{\text{def}}{=} \bigcup_{\rho \in \mathsf{runs}(K)} \mathcal{L}(\rho)$ and $\omega\text{-}\mathsf{traces}(K) \stackrel{\text{def}}{=} \bigcup_{\rho \in \omega\text{-}\mathsf{runs}(K)} \mathcal{L}(\rho)$ the sets of finite and infinite traces of K. Given a propositional sequence σ, and a subset of propositions $P \subseteq \mathbb{P}$, the *restriction* of σ to P is a propositional sequence, noted $\sigma_{\searrow P}$, such that $\forall i \in [0, |\sigma|) : \sigma_{\searrow P}(i) = \sigma(i) \cap P$. Restriction is extended to sets of propositional sequences as expected. Given a set $S \subseteq (2^{\mathbb{P}})^* \cup (2^{\mathbb{P}})^\omega$, $S_{\searrow P} \stackrel{\text{def}}{=} \bigcup_{\sigma \in S} \sigma_{\searrow P}$.

It is often useful to compare two KSs. In the literature, numerous relations have been defined. One of the most common is simulation [Milner, 1980].

Definition 1.8 (Simulation Relation)

Given two KSs $K_1 = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1 \rangle$, $K_2 = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2 \rangle$ over the same set of propositions $\mathbb{P}$, a binary relation $S \subseteq Q_1 \times Q_2$ is a simulation relation between K_1 and K_2, noted $K_1 \trianglelefteq_S K_2$, if the following holds:

(i) $\forall q_1 \in Q_1, \forall q_2 \in Q_2 : \langle q_1, q_2 \rangle \in S \Rightarrow \mathcal{L}_1(q_1) = \mathcal{L}_2(q_2)$

(ii) $\forall q_1 \in I_1, \exists q_2 \in I_2 : \langle q_1, q_2 \rangle \in S$

(iii) $\forall q_1, q_1' \in Q_1, \forall q_2 \in Q_2 : \left(\begin{array}{c} \langle q_1, q_2 \rangle \in S \wedge q_1 \rightarrow_1 q_1' \\ \Rightarrow \\ \exists q_2' \in Q_2 : (q_2 \rightarrow_2 q_2') \wedge \langle q_1', q_2' \rangle \in S \end{array} \right)$

$K_1 = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1 \rangle$ is simulated by $K_2 = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2 \rangle$ (or equivalently K_2 simulates K_1), noted $K_1 \trianglelefteq K_2$, if there exists a relation $S \subseteq Q_1 \times Q_2$ such that $K_1 \trianglelefteq_S K_2$. Finally, K_1 and K_2 are *bisimilar*, noted $K_1 \bowtie K_2$, if there exists a relation $S \subseteq Q_1 \times Q_2$ such that $K_1 \trianglelefteq_S K_2$ and $K_2 \trianglelefteq_{S^{-1}} K_1$. Simulation has several nice properties. For instance, it implies trace inclusion, as formalized hereafter.

Theorem 1.6 (Simulation and Traces)

Given two KSs K_1, K_2 over the same set of propositions $\mathbb{P}$, we have that:

$$(K_1 \trianglelefteq K_2) \Rightarrow (\omega\text{-}\mathsf{traces}(K_1) \subseteq \omega\text{-}\mathsf{traces}(K_2))$$

Simulation can sometimes be too restrictive. Indeed it requires that every move of K_1 is mimicked by *one* move in K_2. When dealing with systems of different granularity, however, it is often necessary to simulate one move of K_1 with *several* consecutive moves of K_2. This is captured by the notion of *stuttering simulation* [Browne et al., 1988]. This is based on the underlying notion of *stuttering transition*. Intuitively, a stuttering transition is a transition that cannot be, in some sense, observed from the outside world, from a propositional perspective, i.e. the transition does not change the value of any proposition. Given the transition relation $\rightarrow$ of a KS K, we define the corresponding *stuttering transition* relation as $\stackrel{\sim}{\rightarrow} \stackrel{\mathbf{def}}{=} \{\langle q, q'\rangle \in Q \times Q \mid q \rightarrow q' \wedge \mathcal{L}(q) = \mathcal{L}(q')\}$. We also define the reflexive and transitive closure of the stuttering transition relation as $\stackrel{\sim}{\rightarrow} \stackrel{\mathbf{def}}{=} (\stackrel{\sim}{\rightarrow})^*$. This directly leads us to the definition of stuttering simulation. Intuitively, as far as stuttering simulation is considered, one transition of K_1 can be mimicked by a certain number of stuttering transitions (possibly 0) followed by one traditional transition of K_2. The definition follows.

Definition 1.9 (Stuttering Simulation Relation)

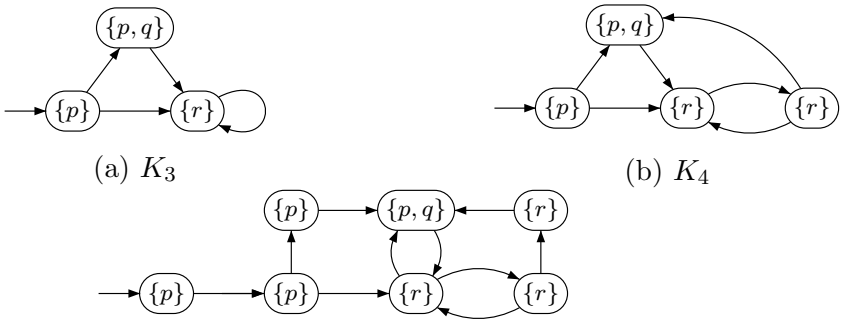
Given two KS $K_1 = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1\rangle$, $K_2 = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2\rangle$ over the same set of propositions $\mathbb{P}$, a binary relation $S \subseteq Q_1 \times Q_2$ is a stuttering simulation relation between K_1 and K_2, noted $K_1 \lesssim_S K_2$ if the following holds:

(i) $\forall q_1 \in Q_1, \forall q_2 \in Q_2 : \langle q_1, q_2\rangle \in S \Rightarrow \mathcal{L}_1(q_1) = \mathcal{L}_2(q_2)$

(ii) $\forall q_1 \in I_1, \exists q_2 \in I_2 : \langle q_1, q_2\rangle \in S$

(iii) $\forall q_1, q_1' \in Q_1, \forall q_2 \in Q_2 : \left(\begin{array}{c} \langle q_1, q_2\rangle \in S \wedge q_1 \rightarrow_1 q_1' \\ \Rightarrow \\ \exists q_2', q_2'' \in Q_2 : (q_2 \stackrel{\sim}{\rightarrow}_2 q_2'' \rightarrow_2 q_2') \wedge \langle q_1', q_2'\rangle \in S \end{array} \right)$

$K_1 = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1\rangle$ is stuttering simulated by $K_2 = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2\rangle$ (or equivalently K_2 stuttering simulates K_1), noted $K_1 \lesssim K_2$, if there exists a relation $S \subseteq Q_1 \times Q_2$ such that $K_1 \lesssim_S K_2$. Bisimilarity is lifted to stuttering bisimilarity as expected: K_1 and K_2 are *stuttering bisimilar*, noted $K_1 \bowtie K_2$, if there exists a relation $S \subseteq Q_1 \times Q_2$ such that $K_1 \lesssim_S K_2$ and $K_2 \lesssim_{S^{-1}} K_1$.



(a) K_3 (b) K_4

(c) K_5

Figure 1.5 - Example of (stuttering) simulation

Example 1.7

Figure 1.5 shows three KSs K_3 and K_4, and K_5. The first two, K_3 and K_4 are related by simulation, i.e. $K_3 \trianglelefteq K_4$. On the other hand, K_3 and K_5 are not. Indeed, from the initial state of K_3, a transition can be taken to a state where r holds. This is clearly not the case in K_5. However, K_3 and K_5 are related by stuttering simulation, i.e. $K_3 \trianglelefteqslant K_5$.

Although not as restrictive as simulation, stuttering simulation still enjoys some interesting properties. The most remarkable is that if $K_1 \trianglelefteqslant K_2$, then to every trace of K_1 corresponds a trace of K_2 that is equivalent modulo stuttering. This equivalence is called *stuttering equivalence*. Intuitively two propositional sequences are stuttering equivalent if they differ only in the number of times each set of propositions consecutively repeats. Formally, given a set of propositions $\mathbb{P}$, two propositional sequences $\sigma_1, \sigma_2 \in (2^{\mathbb{P}})^\omega$ are stuttering equivalent, noted $\sigma \simeq \sigma'$ if and only if there exists two infinite strictly increasing sequences of naturals $i_0\, i_1\, \ldots$ and $j_0\, j_1\, \ldots$ with $i_0 = j_0 = 0$ such that $\forall k \in \mathbb{N}$:

$$\sigma(i_k) = \sigma(i_k + 1) = \ldots = \sigma(i_{k+1} - 1) = \sigma'(j_k) = \sigma'(j_k + 1) = \ldots = \sigma'(j_{k+1} - 1)$$

We then have the following result.

Theorem 1.7 (Stuttering Simulation and Traces)

Given two KSs K_1, K_2 over the same set of propositions $\mathbb{P}$, we have that:

$$(K_1 \trianglelefteqslant K_2) \Rightarrow (\forall \sigma_1 \in \omega\text{-}\mathsf{traces}(K_1), \exists \sigma_2 \in \omega\text{-}\mathsf{traces}(K_2) : \sigma_1 \simeq \sigma_2)$$

1.5.2 Linear Temporal Logic

The first logic we study is the Linear Temporal Logic (LTL), first introduced by Pnueli [Pnueli, 1977]. In LTL, time is assumed to have a *linear* structure. Formulae in LTL are defined using the syntax presented hereafter.

Definition 1.10 (Syntax of Linear Temporal Logic)

Given a set of propositions $\mathbb{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

where $p \in \mathbb{P}$.

In the previous definition, X and U are *temporal modalities*, where X stands for *next* and U for *until*. Other traditional Boolean constructs ($\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\bot$) are derived as in the propositional case. Moreover, on top of the core syntax presented above, one can also define some other derived modalities. First, the F modality, standing for *finally*, is defined as $\mathsf{F}\,\varphi \stackrel{\text{def}}{=} \top\,\mathsf{U}\,\varphi$. Then, the G modality, standing for *globally* is defined as $\mathsf{G}\,\varphi \stackrel{\text{def}}{=} \neg\mathsf{F}\,\neg\varphi$. The size of a LTL formula φ, noted $|\varphi|$ is defined as the number of propositions, operators and modalities appearing in φ. LTL formulae are interpreted over infinite propositional sequences. The semantics is defined by the satisfaction relation.

Definition 1.11 (Semantics of LTL)

Given a set of propositions $\mathbb{P}$, let φ, ψ be two LTL formulae over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition, $\sigma \in (2^{\mathbb{P}})^{\omega}$ be a propositional sequence over $\mathbb{P}$ and $i \in \mathbb{N}$ be a natural. The satisfaction relation for LTL, noted $\models_{\mathsf{L}}$, is defined inductively as follows:

$$\begin{array}{lll}
\langle\sigma, i\rangle \models_{\mathsf{L}} \top & & \\
\langle\sigma, i\rangle \models_{\mathsf{L}} p & \text{iff} & p \in \sigma(i) \\
\langle\sigma, i\rangle \models_{\mathsf{L}} \neg\varphi & \text{iff} & \langle\sigma, i\rangle \not\models_{\mathsf{L}} \varphi \\
\langle\sigma, i\rangle \models_{\mathsf{L}} \varphi \vee \psi & \text{iff} & \langle\sigma, i\rangle \models_{\mathsf{L}} \varphi \text{ or } \langle\sigma, i\rangle \models_{\mathsf{L}} \psi \\
\langle\sigma, i\rangle \models_{\mathsf{L}} \mathsf{X}\,\varphi & \text{iff} & \langle\sigma, i+1\rangle \models_{\mathsf{L}} \varphi \\
\langle\sigma, i\rangle \models_{\mathsf{L}} \varphi\,\mathsf{U}\,\psi & \text{iff} & \text{there exists } k \in [i, +\infty) \text{ such that } \langle\sigma, k\rangle \models_{\mathsf{L}} \varphi \\
 & & \text{and for all } j \in [i, k),\ \langle\sigma, j\rangle \models_{\mathsf{L}} \psi
\end{array}$$

Given a LTL formula φ, we note $[\![\varphi]\!]_{\mathsf{L}} \stackrel{\text{def}}{=} \{\sigma \in (2^{\mathbb{P}})^{+} \mid \langle\sigma, 0\rangle \models_{\mathsf{L}} \varphi\}$. A KS $K = \langle Q, I, \mathcal{L}, \rightarrow\rangle$ satisfies a LTL formula φ, noted $K \models_{\mathsf{L}} \varphi$, if and only if $\omega\text{-}\mathsf{traces}(K_T) \subseteq [\![\varphi]\!]_{\mathsf{L}}$. In other words, K satisfy φ if every infinite trace of K satisfies φ. The LTL model checking problem (LTL-MC) consists in determining if a given Kripke structure satisfies

a LTL formula φ. For finite-state KSs, this problem is known to be PSPACE-complete.

Theorem 1.8 (Complexity of LTL-MC [Sistla and Clarke, 1985])

The LTL model checking problem is PSPACE-complete for finite Kripke structures.

The LTL model checking is classically solved using the automata-based approach [Vardi and Wolper, 1986], whereby the LTL formula φ is used to build a Büchi automaton A_φ accepting exactly $[\![\varphi]\!]_{\mathsf{L}}$. Then, for a given KS K, the model checking problem can be reduced to check if $\omega\text{-}\mathsf{traces}(K) \subseteq \omega\text{-}\mathsf{lang}(A_\varphi)$, or equivalently if $\omega\text{-}\mathsf{traces}(K) \cap \overline{\omega\text{-}\mathsf{lang}(A_\varphi)} = \varnothing$. In practice, since complementing Büchi automata is hard (see e.g. [Vardi, 2007]), instead of computing A_φ and then complementing it, it is preferable to compute $A_{\neg\varphi}$ directly, which is equivalent. With this technique, the LTL model checking problem boils down to checking if $\omega\text{-}\mathsf{traces}(K) \cap \omega\text{-}\mathsf{lang}(A_{\neg\varphi}) = \varnothing$. This technique has been continuously refined over the years. See for instance [Vardi, 1995; Raskin and Doyen, 2007].

Example 1.8

Let us give a few examples of LTL formulae and illustrate their semantics with some infinite propositional sequences. In these few examples, we note "..." the infinite repetition of the last set of propositions.

- The formula $\mathsf{X}(p \wedge q)$ states that in the second position of the sequence, p and q should hold. The sequence $\{r\}, \{\underline{p}, \underline{q}, r\}, \{p\}, \varnothing, \{q\}, \ldots$ is a model for this formula. On the other hand, the formula $\{p\}, \{\underline{q}\}, \{p\}, \{q\}, \ldots$ is not.
- The formula $\mathsf{G}\, p$ states that in every positions of the sequence, p should hold. The sequence $\{\underline{p}, q\}, \{\underline{p}, r\}, \{\underline{p}\}, \{\underline{p}, r\}, \ldots$ is a model for this formula. On the other hand, the sequence $\{p, q\}, \{\underline{q}\}, \{p\}, \ldots$ is not.
- The formula $\mathsf{F}(r \wedge s)$ states that there should exist a position in the sequence where r and s hold. The sequence $\{s\}, \{p, q\}, \{\underline{r}, \underline{s}\}, \{p, s\}, \ldots$ is a model for this formula. On the other hand the sequence $\{\underline{s}\}, \{\underline{p, q, r}\}, \{\underline{p, q, s}\}, \ldots$ is not.

A particular interest has been found for a subset of LTL where the X modality is forbidden. Indeed, this subset, noted LTL-X in the following, has the remarkable property that it is closed under stuttering equivalence.

Theorem 1.9 (LTL-X is Stuttering-Closed [Lamport, 1983])

Given a set of proposition $\mathbb{P}$, let φ be a LTL-X formula over $\mathbb{P}$ and $\sigma, \sigma' \in (2^{\mathbb{P}})^\omega$ be two infinite propositional sequences such that $\sigma \simeq \sigma'$. We have that:

$$(\, \langle \sigma, 0 \rangle \models_{\mathsf{L}} \varphi \,) \Leftrightarrow (\, \langle \sigma', 0 \rangle \models_{\mathsf{L}} \varphi \,)$$

This implies the following result.

Theorem 1.10 (Stuttering Simulation Preserves LTL-X)

Given a set of propositions $\mathbb{P}$, let K_1, K_2 be two KSs over $\mathbb{P}$ such that $K_1 \trianglelefteq K_2$ and φ be a LTL-X formula over $\mathbb{P}$. We have that:

$$(K_2 \models_{\mathsf{L}} \varphi) \Rightarrow (K_1 \models_{\mathsf{L}} \varphi)$$

Proof

By Theorem 1.7, we have that $\forall \sigma_1 \in \omega\text{-}\mathsf{traces}(K_1), \exists \sigma_2 \in \omega\text{-}\mathsf{traces}(K_2) : \sigma_1 \simeq \sigma_2$. By Theorem 1.9, we know that $\langle \sigma_1, 0 \rangle \models_{\mathsf{L}} \varphi$ holds iff $\langle \sigma_2, 0 \rangle \models_{\mathsf{L}} \varphi$. However, by hypothesis, we know that $K_2 \models_{\mathsf{L}} \varphi$, which implies that $\langle \sigma_2, 0 \rangle \models_{\mathsf{L}} \varphi$. We can conclude that $\forall \sigma_1 \in \omega\text{-}\mathsf{traces}(K_1) : \langle \sigma_1, 0 \rangle \models_{\mathsf{L}} \varphi$ which implies that $K_1 \models_{\mathsf{L}} \varphi$.

1.5.3 Computational Tree Logic

The second temporal logic that we study is the Computational Tree Logic (CTL), first introduced by Clarke and Emerson [Clarke and Emerson, 1981]. A similar logic was also studied independently by Sifakis and Queille [Queille and Sifakis, 1982]. In CTL, time is assumed to have a *branching* structure. Formulae in CTL are defined using the syntax presented hereafter.

Definition 1.12 (Syntax of Computational Tree Logic)

Given a set of proposition $\mathbb{P}$, a formula in Computational Tree Logic (CTL) is defined using the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{AX}\,\varphi \mid \mathsf{E}(\varphi\,\mathsf{U}\,\varphi) \mid \mathsf{A}(\varphi\,\mathsf{U}\,\varphi)$$

where $\in \mathbb{P}$.

In the previous definition, EX, AX, EU, and AU are *temporal modalities*, where X and U stands respectively for *next* and *until* (as in LTL), and where E stands for *exists a path*, and A for *for all paths*. Other traditional Boolean constructs ($\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\bot$) are derived as in the propositional case. Similarly to LTL, one can also define some derived modalities. First, the *finally* modalities EF and AF are defined respectively as $\mathsf{EF}\,\varphi \stackrel{\mathbf{def}}{=} \mathsf{E}(\top\,\mathsf{U}\,\varphi)$ and $\mathsf{AF}\,\varphi \stackrel{\mathbf{def}}{=} \mathsf{A}(\top\,\mathsf{U}\,\varphi)$. Then, the *globally* modalities EG and AG are defined respectively as $\mathsf{EG}\,\varphi \stackrel{\mathbf{def}}{=} \neg\mathsf{AF}\,\neg\varphi$ and $\mathsf{AG}\,\varphi \stackrel{\mathbf{def}}{=} \neg\mathsf{EF}\,\neg\varphi$. Similarily to LTL, the size of a CTL formula φ, noted $|\varphi|$, is defined as the number of propositions, operators and modalities appearing in φ. CTL formulae are interpreted over states of a Kripke structure. The semantics is defined by the satisfaction relation.

Definition 1.13 (Semantics of CTL)

Given a set of propositions $\mathbb{P}$, let φ, ψ be two CTL formulae over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition, $K = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ be a KS, and $q \in Q$ be a state of K. The satisfaction relation for CTL, noted $\models_{\mathsf{C}}$, is defined inductively as follows:

$$
\begin{array}{lll}
\langle K, q \rangle \models_{\mathsf{C}} \top & & \\
\langle K, q \rangle \models_{\mathsf{C}} p & \text{iff} & p \in \mathcal{L}(q) \\
\langle K, q \rangle \models_{\mathsf{C}} \neg\varphi & \text{iff} & \langle K, q \rangle \not\models_{\mathsf{C}} \varphi \\
\langle K, q \rangle \models_{\mathsf{C}} \varphi \vee \psi & \text{iff} & \langle K, q \rangle \models_{\mathsf{C}} \varphi \text{ or } \langle K, q \rangle \models_{\mathsf{C}} \psi \\
\langle K, q \rangle \models_{\mathsf{C}} \mathsf{EX}\,\varphi & \text{iff} & \text{there exists } q' \in \mathsf{post}(q) \text{ such that } \langle K, q' \rangle \models_{\mathsf{C}} \psi \\
\langle K, q \rangle \models_{\mathsf{C}} \mathsf{AX}\,\varphi & \text{iff} & \text{for all } q' \in \mathsf{post}(q),\ \langle K, q' \rangle \models_{\mathsf{C}} \psi \\
\langle K, q \rangle \models_{\mathsf{C}} \mathsf{E}(\varphi\,\mathsf{U}\,\psi) & \text{iff} & \text{there exists } \rho \in \omega\text{-}\mathsf{runs}(q) \text{ and } k \in \mathbb{N} \text{ such that} \\
 & & \langle K, \rho(k) \rangle \models_{\mathsf{C}} \psi \text{ and for all } j \in [0, k),\ \langle K, \rho(j) \rangle \models_{\mathsf{C}} \varphi \\
\langle K, q \rangle \models_{\mathsf{C}} \mathsf{A}(\varphi\,\mathsf{U}\,\psi) & \text{iff} & \text{for all } \rho \in \omega\text{-}\mathsf{runs}(q), \text{there exists } k \in \mathbb{N} \text{ such that} \\
 & & \langle K, \rho(k) \rangle \models_{\mathsf{C}} \psi \text{ and for all } j \in [0, k),\ \langle K, \rho(j) \rangle \models_{\mathsf{C}} \varphi
\end{array}
$$

Given a KS $K = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a CTL formula φ, we note $[\![\varphi]\!]^K_{\mathsf{C}} \stackrel{\text{def}}{=} \{q \in Q \mid \langle K, q \rangle \models_{\mathsf{C}} \varphi\}$. A KS satisfies a CTL formula φ, noted $K \models_{\mathsf{C}} \varphi$, if and only if $I \subseteq [\![\varphi]\!]^K_{\mathsf{C}}$. In other words, K satisfies φ if φ holds in every initial state of K. The CTL model checking problem (CTL-MC) consists in determining if a given Kripke Structure satisfies a CTL formula. For finite-state KSs, this problem is known to be PTIME-complete.

Theorem 1.11 (Complexity of CTL-MC [Clarke et al., 1983])

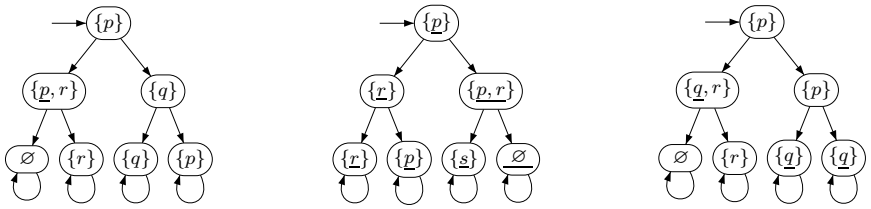
The CTL model checking problem is PTIME-complete for finite Kripke structures.

The model checking for CTL is classically solved using the symbolic approach [McMillan, 1993]. In this approach, given a KS K and CTL formula φ, the set of states where φ holds, i.e. $[\![\varphi]\!]^K_{\mathsf{C}}$, is computed symbolically using efficient data structures like e.g. BDD [Bryant, 1992]. This computation is done inductively on the structure of φ. For propositions and Boolean operator, this is done by examining K. For temporal modalities, this is done by exploiting the fixed point characterization of CTL, formalized hereafter.

Theorem 1.12 (Fixed Point Characterization of CTL [Emerson and Clarke, 1980])

Given a KS $K = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and two CTL formulae φ and ψ, we have that:

$$
\begin{array}{rcl}
[\![\mathsf{EX}\,\varphi]\!]^K_{\mathsf{C}} & = & \mathsf{pre}([\![\varphi]\!]^K_{\mathsf{C}}) \\
[\![\mathsf{AX}\,\varphi]\!]^K_{\mathsf{C}} & = & \widetilde{\mathsf{pre}}([\![\varphi]\!]^K_{\mathsf{C}}) \\
[\![\mathsf{A}(\varphi\,\mathsf{U}\,\psi)]\!]^K_{\mathsf{C}} & = & \mathsf{lfp}\ \lambda X \cdot [\![\psi]\!]^K_{\mathsf{C}} \cup ([\![\varphi]\!]^K_{\mathsf{C}} \cap \widetilde{\mathsf{pre}}(X)) \\
[\![\mathsf{E}(\varphi\,\mathsf{U}\,\psi)]\!]^K_{\mathsf{C}} & = & \mathsf{lfp}\ \lambda X \cdot [\![\psi]\!]^K_{\mathsf{C}} \cup ([\![\varphi]\!]^K_{\mathsf{C}} \cap \mathsf{pre}(X))
\end{array}
$$



(a) $K_6 \models_{\mathsf{C}} \mathsf{EX}\, p$ (b) $K_7 \models_{\mathsf{C}} \mathsf{AG}\, \neg q$ (c) $K_8 \models_{\mathsf{C}} \mathsf{AF}\, q$

(d) $K_9 \not\models_{\mathsf{C}} \mathsf{EX}\, p$ (e) $K_{10} \not\models_{\mathsf{C}} \mathsf{AG}\, \neg q$ (f) $K_{11} \not\models_{\mathsf{C}} \mathsf{AF}\, q$

Figure 1.6 - Example of KSs satisfying, or not, some CTL formulae

Once $[\![\varphi]\!]^K_{\mathsf{C}}$ is computed, one can easily check that $I \subseteq [\![\varphi]\!]^K_{\mathsf{C}}$.

Example 1.9

Let us give a few examples of formulae and illustrate their semantics with some KSs. We chose tree-like KSs, in order to make the (non-)satisfaction explicitly apparent.

- The formula $\mathsf{EX}\, p$ states that every initial state should have a successor state where p holds. The KS K_6 presented in Figure 1.6 (a) is a model for this formula. On the other hand, the KS K_9 of Figure 1.6 (d) is not.
- The formula $\mathsf{AG}\, \neg q$ states that in all states reachable from an initial state, q should not hold. The KS K_7 presented in Figure 1.6 (b) is a model for this formula. On the other hand, the KS K_{10} of Figure 1.6 (e) is not.
- The formula $\mathsf{AF}\, q$ states that in every run of the KS, a state where q holds should be eventually reached. The KS K_8 presented in Figure 1.6 (c) is a model for this formula. On the other hand, the KS K_{11} of Figure 1.6 (f) is not.

Chapter

2

Distributed Supervision Language

« *There is no programming language, no matter how structured, that will prevent programmers from writing bad programs.* »

Larry Flon

The *Distributed Supervision Language* (dSL) is a full-featured programming language and design environment dedicated to the development of distributed industrial control applications. dSL was created at *Macq Electronique*[1], in collaboration with the *Université Libre de Bruxelles* [De Wachter et al., 2003a; De Wachter et al., 2003b; De Wachter et al., 2005; De Wachter, 2005]. It is the successor of the *Supervision Language* (SL) which was also created at *Macq Electronique*. As illustrated in Figure 2.1, a system in SL is composed of several entities communicating with one another through a network:

- the *Programmable Logic Controllers* (PLCs) are the entities interfaced, through sensors and actuators, with the environment. Note that each PLC has only a partial/local view of the environment. PLCs are programmed using a low-level assembler-like language. For a complete technical description, see [Macq Electronique, 2006];
- the *Supervisor* is the interface between the system and its administrator(s), typically through a graphical user interface. Its main job is to collect data from the PLCs in order to construct a global view of the entire environment. From that global view, the supervisor can determine the appropriate actions to be taken,

[1] a Belgian company specialized in industrial process control `http://www.macqel.be`

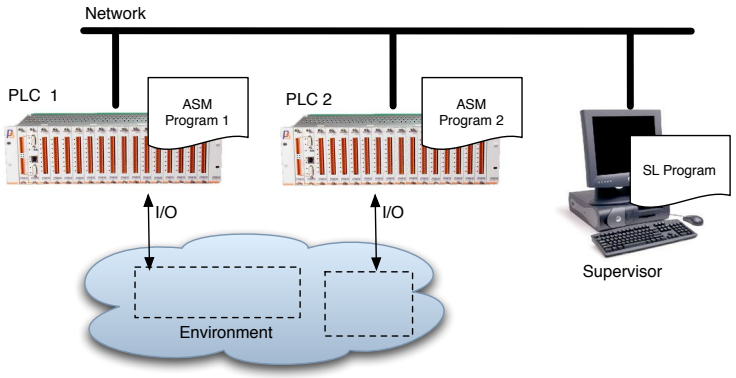


Figure 2.1 - Overview of a SL system

which are then forwarded to the PLCs. The supervisor is programmed using the SL language.

This approach suffers from several drawbacks. First of all, two separate languages have to be used (and learned by the programmers) in order to develop such a system: the assembler-like language for PLCs and SL for the supervisor. Moreover, the communications between the PLCs and the supervisor must be specified explicitly in each program of the system. This renders the development of a system with this approach quite tedious and, more importantly, error-prone.

The dSL language was introduced precisely to remedy those problems. First, dSL was designed to be compiled into a low-level language interpreted by a virtual machine. An implementation of this virtual machine is available on both the PLCs and the supervisor, thus allowing dSL to be used as a common programming language for the whole system. But, most importantly, the main advantage of dSL is its transparent and automatic distribution. In the dSL approach, an application is programmed as if the whole system were centralized, allowing the programmer to access transparently all sensors and actuators of the system as if they were localised on a single execution site. Then, as illustrated in Figure 2.2, a dSL program is compiled and automatically distributed. The distribution process is parametrized by the locations of all inputs and outputs of the system, i.e. to which PLC is connected what sensor/actuator. This information, provided in the *Localization Database*, allows the compiler/distributor to generate several pieces of code, each intended for one entity of the systems (PLC or Supervisor). Furthermore, communication stubs are added automatically into each piece of code so that the overall behaviour of the system corresponds to the original

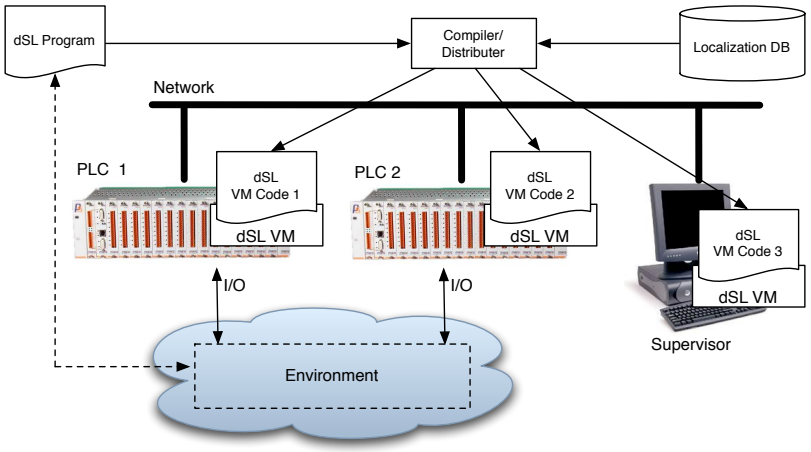


Figure 2.2 - Overview of a dSL system

program. This allows the programmer to focus on the functional aspects of the program, while leaving most of the technical details to the compiler. This approach also leads to more maintainable code. Indeed, since the distribution is done automatically, a change in the configuration of sensors and/or actuators only implies the program to be recompiled[2], after the update.

In the remainder, we detail the dSL language. First, in Section 2.1, we present and illustrate the syntax of the language. Next, in Section 2.2 we explain how a dSL program is distributed. Finally, we come to a conclusion in Section 2.3, by discussing related works. The content of this chapter is based on a joint work with Bram De Wachter and Thierry Massart, published in [De Wachter et al., 2003a] and in [De Wachter et al., 2003b].

2.1 Syntax

dSL is an imperative language with static variables, i.e. no dynamic memory allocation. Global variables can be linked to sensors and actuators allowing the program to interact with its environment. Moreover, in keeping with the paradigm of its ancestor SL, dSL is *event-driven*. This paradigm allows the programmer to express that some piece of code should be executed every time a particular condition is met. dSL also offers limited object-oriented [Abadi and Cardelli, 1996; Rumbaugh et al., 1991] features, with all the benefits they may provide.

[2]however, we will see later on that some minor modifications of the code may be required

Note that the syntax presented in this section slightly differs from the one used at *Macq Electronique*. Indeed, in their development environment, sites, classes and global variables are declared in the *localization database*, not in the program itself. In the parser developed at the *Université Libre de Bruxelles*, for simplicity reasons, we chose to include them in the program itself. We therefore introduced some new constructs, in the same spirit as the existing syntax, in order to deal with those declarations. The full grammar of this dialect of dSL can be found in Appendix A.1.

2.1.1 A Running Example

In order to get the reader acquainted with the syntax of dSL, let us introduce a small example program that is used throughout this section to illustrate the various constructs of the language.

Example 2.1

The program presented in Figure 2.3 is used to control a boiler. This boiler is composed of a tank that needs to be filled up with water. This water is then brought to the boil using a gas burner. The tank is equipped with two sensors: a temperature sensor and a water level sensor. The temperature sensor is used to detect when the water is boiling or when it is too cold, in which case the appropriate action needs to be taken. The water level sensor is used to detect when the level of water is too low, in which case the valve for the water intake must be opened to fill up the tank again. In addition, the system includes a control panel with two indicators. The first one indicates when the boiler is ready. The second one is used to indicated that a maintenance team should be alerted to check the gas burner. Furthermore, the fact that the temperature in the tank is below zero when the controller is started is logged when the program is started for later use by the maintenance team.

As demonstrated in Example 2.1, a program in dSL is composed of six parts: *types*, *global variables*, *sites*, *methods*, *events* and *sequences* declarations. Let us review and explain each of these parts, and detail their syntax.

2.1.2 Types

dSL supports several basic data types such as booleans (`BOOL`), integers (`INT` or `LONG`) and reals (`REAL`). In addition to basic data types, dSL allows to define classes (sometimes called structured types) grouping several fields together. Classes can be defined with the keyword `CLASS`, using the syntax presented in Figure 2.4.

```
 1 (* Types *)
 2 CLASS TANK
 3   level, temperature : INT;
 4   valve              : BOOL;
 5 END_CLASS
 6
 7 CLASS BURNER
 8   state              : BOOL;
 9   usage              : INT;
10 END_CLASS
11
12 (* Global variables *)
13 VAR
14   tank               : TANK;
15   burner             : BURNER;
16   maintenance, ready : BOOL;
17   frozen             : BOOL;
18 END_VAR
19
20 (* Sites *)
21 SITE boilerRoom
22   INPUT tank.temperature : 1.0.1;
23   INPUT tank.level       : 1.0.2;
24   OUTPUT tank.valve      : 1.2.1;
25   OUTPUT burner.state    : 1.2.2;
26 END_SITE
27
28 SITE controlRoom
29   OUTPUT maintenance     : 3.0.1;
30   OUTPUT ready           : 3.0.2;
31 END_SITE
32
33 (* Methods *)
34 METHOD BURNER::turnOn()
35   self.state := TRUE;
36   self.usage := self.usage + 1;
37 END_METHOD
38
39 METHOD BURNER::turnOff()
40   self.state := FALSE;
41 END_METHOD
43 METHOD TANK::openValve()
44   self.valve := TRUE;
45 END_METHOD
46
47 METHOD TANK::closeValve()
48   self.valve := FALSE;
49 END_METHOD
50
51 (* Events *)
52 WHEN tank.temperature < 80 THEN
53   burner<-turnOn();
54 END_WHEN
55
56 WHEN tank.temperature > 100 THEN
57   burner<-turnOff();
58 END_WHEN
59
60 WHEN ~burner.usage > 100000 THEN
61   maintenance := TRUE;
62 END_WHEN
63
64 WHEN IN TANK self.level < 8 THEN
65   self<-openValve();
66 END_WHEN
67
68 WHEN IN TANK self.level > 10 THEN
69   self<-closeValve();
70 END_WHEN
71
72 (* Sequences *)
73 SEQUENCE main()
74   IF tank.temperature < 0 THEN
75     frozen := TRUE;
76   END_IF;
77   WAIT(tank.level > 10 AND
78        tank.temperature > 100);
79   ready := TRUE;
80 END_SEQUENCE
```

Figure 2.3 - A boiler control program

⟨class_decl⟩ ::= CLASS ⟨id⟩
⟨id_list⟩ : ⟨type⟩ ;
...
⟨id_list⟩ : ⟨type⟩ ;
END_CLASS

⟨type⟩ ::= BOOL | INT | LONG | REAL | ⟨id⟩

Figure 2.4 - Class declaration

Example 2.2

In the program of Figure 2.3, two classes are declared at lines 2–10. The first is the class `TANK`, at lines 2–5, used to model the tank with three fields. The first two fields are `level` and `temperature` which are used for the two sensors in the tank, and the third field is `valve`, that is used as the command of the valve for the water intake of the tank. The second class is the class `BURNER`, at lines 7–10, used to model the gas burner, composed of two fields. The field `state` models the state of the heater (on/off) and the field `usage` is used to count the number of times the burner is turned on.

2.1.3 Global Variables

As in any other programming language, global variables are variables visible throughout the entire program. dSL is no exception. The localization of all global variables is statically decided at compile-time. Once assigned to a given execution site (PLC or supervisor), a global variable cannot move. As we shall see in the following sections, this implies some restrictions on event-driven code. Global variables can be declared with the keyword `VAR` using the syntax presented in Figure 2.5.

Example 2.3

In the program of Figure 2.3, five global variables are declared at lines 13–18. The first two variables, `tank` and `burner` at lines 14–15, are used to model the boiler. The next two boolean variables, `maintenance` and `ready` at line 16, are used to control the led on the control panel. Finally the variable `frozen` at line 17, is used to log the fact that the water in the tank was frozen when the boiler started.

⟨var_decl⟩ ::= `VAR`
⟨id_list⟩ : ⟨type⟩ ;
...
⟨id_list⟩ : ⟨type⟩ ;
`END_VAR`

Figure 2.5 - Global variable declaration

2.1.4 Sites

In dSL, global variables can be either *internal* or *external*. As their name indicates, internal variables are only used internally in the program like traditional global variables in other programming languages. External variables, on the other hand, are attached to physical devices (sensors or actuators). Variables attached to sensors are called input variables. During the execution of the program, the values of those input variables are periodically updated with the state of the sensors they are attached to. Variables attached to actuators are called output variables. Similarly to input variables, the values of those output variables are periodically used to update the states of the actuators they are attached to.

External variables need to be declared like any other global variables, as explained in the previous section. Then each variable needs to be attached to a physical device. In dSL, each device is identified by the PLC to which it is connected (its execution site) and a three digit address, e.g. `1.2.3`. In such an address, the first number identifies the rack in the PLC, the second number identifies the I/O card in this rack and the third identifies the slot in this card. Each input, respectively output, variable is specified using the keyword `INPUT`, respectively `OUTPUT`, along with the address of the device on the PLC. Finally, those declarations are grouped into execution sites, with the keyword `SITE`, using the syntax presented in Figure 2.6.

Example 2.4

In the program of Figure 2.3, two sites are declared at lines 21–31. The first is the `boilerRoom` site at lines 21–26, where the tank and the gas burner are located. The other is the `controlRoom` site, at lines 28–31, where the control panel is located.

2.1.5 Methods

Methods are declared over previously defined classes. Methods can be parametrized. Parameters are evaluated in a *call-by-value* fashion (like in C). Local variables can be

⟨site_decl⟩ ::= `SITE` ⟨id⟩
⟨io_type⟩ ⟨lhside⟩ : ⟨nb⟩ . ⟨nb⟩ . ⟨nb⟩ ;
...
⟨io_type⟩ ⟨lhside⟩ : ⟨nb⟩ . ⟨nb⟩ . ⟨nb⟩ ;
`END_SITE`

⟨io_type⟩ ::= `INPUT` | `OUTPUT`

⟨lhside⟩ ::= ⟨id⟩ | ⟨lhside⟩ . ⟨id⟩

Figure 2.6 - Site declaration

declared using the same syntax as for global variables (see Section 2.1.3), only in this case, those variables are only visible inside the body of the method. A special variable `self` is implicitly declared. It can be used to refer to the object on which the method is being called (similar to `this` in `C++`). Methods are declared with the keyword `METHOD` using the syntax presented in Figure 2.7(a).

Similarly to any programming language, method can be called synchronously. In this case, the control is interrupted when the call is made, and resumes when the method has finished its execution. Alternatively, in dSL, methods can be called asynchronously using the keyword `LAUNCH`. In this case, the control returns directly after the call and the method is executed in parallel. In both instances, the call is made with the operator `<-` using the syntax presented in Figure 2.7(b).

Example 2.5

In the program of Figure 2.3, several methods are declared at lines 34–49. The first two are defined on the class `BURNER` at lines 34–41. Those two methods are called respectively at lines 53 and 57 to turn the burner on and off. The last two methods are defined on the class `TANK` at lines 43–49. Those two methods are called respectively at lines 65 and 69 to open and close the valve for the water intake of the tank.

2.1.6 Events

As already mentioned in the introduction, dSL is an *event-driven* language. This allows to express that some piece of code should be executed every time a particular condition is met. It is most important to note that it takes a *rising-edge* of the condition, i.e. the condition must be successively evaluated to ff and then to tt, for an event to be triggered. This event-driven scheme implies that assignments in dSL have side effects.

⟨method_decl⟩ ::= `METHOD` ⟨id⟩ :: ⟨id⟩ `(` ⟨param_decl⟩ `)`
⟨var_decl⟩
⟨block⟩
`END_METHOD`

⟨param_decl⟩ ::= ⟨id_list⟩ `:` ⟨type⟩ `,` ... `,` ⟨id_list⟩ `:` ⟨type⟩ | ε

(a) Method declaration

⟨method_call⟩ ::= `LAUNCH` ⟨id⟩ `<-` ⟨id⟩ `(` ⟨rhside_list⟩ `)`
| ⟨id⟩ `<-` ⟨id⟩ `(` ⟨rhside_list⟩ `)`

⟨rhside_list⟩ ::= ⟨rhside⟩ `,` ... `,` ⟨rhside⟩ | ε

⟨rhside⟩ ::= ⟨lhside⟩ | $\sim$ ⟨lhside⟩ | ⟨constant⟩ | `(` ⟨rhside⟩ `)`
| ⟨un_op⟩ ⟨rhside⟩ | ⟨rhside⟩ ⟨bin_op⟩ ⟨rhside⟩

(b) Method call

Figure 2.7 - Method declaration and call

In fact, every time an assignment is executed, the condition of each event is tested and the corresponding piece of code is executed if necessary. In practice, of course, only events which condition contains the assigned variables are tested. If several events are triggered by the same assignment, those events are treated in the order in which they appear in the program. Note that if an assignment inside an event triggers another event when executed, the first event is interrupted and the second event is executed. This instantaneous triggering scheme may introduce infinite triggering loops that should, of course, be avoided.

As expected, an event is composed of two parts: the condition to be monitored and the piece of code to execute when the condition is met, called the body. Events are declared with the keyword `WHEN` using the syntax presented in Figure 2.8(a).

Furthermore, it is possible to define class events, i.e. events declared for every instance of a particular class. In this case, similarly to methods, a special variable `self` is implicitly declared and used to denote the object on which the event is declared. Note that class events are simply in-lined in the early stage of compilation. Class events are declared using the keywords `WHEN IN`, using the syntax presented in Figure 2.8(b), where ⟨id⟩ identifies the class on which the event is defined.

⟨event_decl⟩ ::= `WHEN` ⟨rhside⟩ `THEN`
⟨block⟩
`END_WHEN`

(a) Simple event

⟨event_decl⟩ ::= `WHEN IN` ⟨id⟩ ⟨rhside⟩ `THEN`
⟨block⟩
`END_WHEN`

(b) Class event

Figure 2.8 - Event declaration

```
1 VAR
2   buttonPressed,
3   lampCommand    : BOOL;
4 END_VAR
5
6 SITE site1
7   INPUT buttonPressed : 1.0.0;
8 END_SITE
10 SITE site2
11   INPUT lampCommand   : 2.0.0;
12 END_SITE
13
14 WHEN buttonPressed THEN
15   lampCommand := TRUE;
16 END_WHEN
```

Figure 2.9 - Example of undistributable program

Example 2.6

In the program of Figure 2.3 several events are defined at lines 52–70. The first two events, at lines 52–58, are used to command the burner. In the first of those two, if the temperature in the tank drops below a certain value, the burner is turned on. The second of those allows to turn it off when the water is boiling. The third event, at lines 60–62, is used to control the maintenance alarm on the control panel. When the burner has been used 100000 times, the alarm is turned on. The last two events, at lines 64–70 are used to control the valve of the tank. When the water level drops below a certain value, the valve is opened. Symmetrically, when the tank is filled up, the valve is closed. Note that the last two events are defined for all instances of the class `TANK`, using the `WHEN IN` construct.

In dSL, the code inside an event (the instructions inside its body as well as the triggering condition) or reachable from an event (through method call) must be *atomic*,

i.e. executed without being interrupted by any communications. The code inside an event, including all global variables used or assigned in this code, must therefore be localised on the same execution site. As illustrated in Example 2.7, this atomicity constraint may lead to undistributable programs.

Example 2.7

Consider the program of Figure 2.9, where a lamp should be turned on when a button is pressed. Intuitively, this program requires that the lamp is turned on *directly* after the button has been pressed. However, in the program, the command for the lamp (`lampCommand`) and the button (`buttonPressed`) are explicitly localised on different execution sites (`site1` and `site2`). A message should be sent from `site1` to `site2`, every time the button is pressed, and this communication could possibly break the atomicity. Therefore, this program cannot be distributed because the atomicity constraint cannot be fulfilled.

In such a case, the programmer still has a way out. If a communication delay is acceptable in a particular case, the programmer may allow it explicitly using the $\sim$ operator. Instead of directly manipulating a variable `x`, the $\sim$ operator allows to handle a local copy of this variable. Then, every time the variable `x` is modified on a remote execution site, that site sends a message indicating that the value of `x` was modified, thus allowing to maintain a *delayed* copy of `x` locally. Note that if several sites manipulate a distant copy of `x`, the message is sent to all of those sites. One should of course be extremely careful when using a *tilded* copy instead of the real variable. Indeed, the value of this copy can be different from the value of the actual variable (this happens when the message is travelling over the network).

Example 2.8

In the program of Figure 2.3, the reader might have noticed the use of the $\sim$ in the event at lines 60–62. Indeed, the atomicity constraint imposed on the event at lines 52–54 implies that the variables `burner.usage` and `burner.state` should be localised on the same execution site (both variables are used in the method `turnOn()` called in the body of this event). On the other hand, the variable `maintainance` corresponding to a led in the control panel, is localised on the site `controlRoom`. This would violate the atomicity constraint on the event at lines 60–62. In this case, however, a delayed copy is still acceptable. Therefore, using the $\sim$ operator at line 60 allows to relax the atomicity constraint, thus allowing the program to be distributed properly.

⟨sequence_decl⟩ ::= `SEQUENCE` ⟨id⟩ `(` ⟨param_decl⟩ `)`
⟨var_decl⟩
⟨block⟩
`END_SEQUENCE`

(a) Sequence declaration

⟨sequence_call⟩ ::= `LAUNCH` ⟨id⟩ `(` ⟨rhside_list⟩ `)`

(b) Sequence call

Figure 2.10 - Sequence declaration and call

2.1.7 Sequences

As their name indicates, sequences are successions of instructions to be executed sequentially. A sequence is essentially an independent process running in parallel with the remainder of the program. In contrast with *event-driven* code, however, sequential code can be *distributed*. In fact, a sequence migrates from sites to sites following the localization of the global variables that are used in each instruction. Each instruction, however, has to be executed on a single execution site, and is consequently atomic. This concept is known as *process* or *thread migration* [Eskicioglu, 1990] whereby, when needed, the execution of a thread is stopped on the current execution site, its context is moved to the next execution site, where it is restored and the execution resumed.

Sequences are declared with the keyword `SEQUENCE` using the syntax presented in Figure 2.10(a). Local variables can be declared inside a sequence. Local variables inside sequences are part of the execution context and therefore move to follow the execution of the sequence. Inside sequences, a special instruction `WAIT` can be used to stop the execution until a particular condition is met. However, in this case, a rising-edge is not required as for event triggering.

Example 2.9

In the program of Figure 2.3, only one sequence is declared at lines 73–80: the sequence `main`. The first part of this sequence, at lines 74–76, checks if the temperature is below zero, in which case, the variable `frozen` is set to `TRUE` to record that fact. Then, at lines 77–78, a `WAIT` instruction is used to wait for the boiler to be ready. As soon as the tank is filled and the water is boiling, the variable `ready` is set to `TRUE`. This turns on the led on the control panel indicating that the boiler is ready to use.

Sequences can only be called asynchronously with the keyword `LAUNCH`. However, contrarily to methods, only one instance of each sequence is allowed to run at any given time. Calls to already running sequences are discarded. Sequences are called using the syntax presented in Figure 2.10(b). Moreover, if a program contains a sequence named `main`, this sequence is implicitly launched when the program is started.

2.2 Distribution

The problem of distributing a dSL program amounts to localising each global variable and each instruction of the program to a given execution site, taking into account the fact that each external variable is already assigned to a particular execution site. This problem can be decomposed in two successive steps. The first step consists in localising the event-driven code by enforcing the atomicity constraints imposed by the program. The second step consists in localising the remaining sequential code. We briefly explain those two successive steps in Section 2.2.1 and Section 2.2.2. For a more complete and formal description of this distribution process, we refer the reader to [De Wachter, 2005, Chapter 4].

2.2.1 Event-Driven Code

The first step in the distribution process is to localise the event-driven code by enforcing the atomicity constraints imposed by the program. The first constraint is that the code inside each event should be atomic, i.e. local to a given execution site. The second constraint is that each sequential instruction should also be atomic. For that purpose, an undirected graph is constructed taking into account those constraints. In this graph, nodes are used to model the events, (untilded) global variables, and instructions of the program. Edges in this graph are added to model the fact that two vertexes should be localised on the same execution site. First, edges are added between variables and the instructions they appear in. Then, edges are added between events and the instructions appearing in them. Finally, edges are added between events and the variables appearing in their conditions. This graph is called the *atomic coloring graph*. Once this graph is built, in order to determine if the program can be distributed, one simply needs to check that there exists no pair of external variables, localised on different execution sites, belonging to the same connected component. Indeed if such a pair of variable exists, it means that, in order to satisfy some atomicity constraints, those two variables should be localised on the same site. However, their localization being explicitly forced in the program to different execution sites makes that impossible.

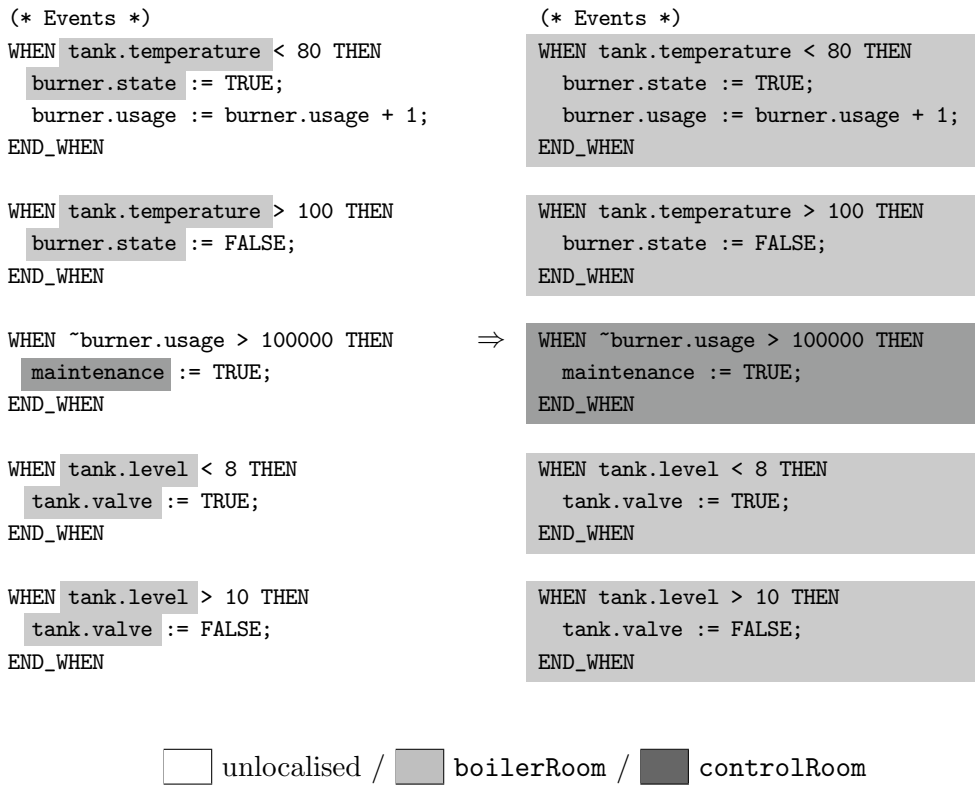


Figure 2.11 - Localising atomic code

We can therefore conclude in this case that the atomicity constraints cannot be satisfied and that the program cannot be distributed. The program is therefore rejected by the compiler/distributor. Otherwise, the localization of external variables given in the program can be propagated throughout each connected component. This can however leave some events unlocalised (events not connected to an external variable). Those events are simply distributed evenly amongst the available execution sites (load balancing).

Example 2.10

Figure 2.11 illustrates the first step on the program of Figure 2.3. To simplify the presentation, methods in the program have been in-lined. In this case, there is at least one external variable in each event. Note that if the $\sim$ was not used in the third event, the program could not be distributed because, both variables `burner.usage` and `maintenance` appear in this event. They would therefore belong to the same connected component.

2.2.2 Sequential Code

The result of the first step is a localization for each event in the program, as well as a localization for some of the sequential instructions. Indeed, all instructions, including sequential instructions, are taken into account in the atomic colouring graph. If an instruction belongs to the connected component of one of the events of the program (because they share a common variable), it will be localised in the first step. The purpose of this second phase is to localise the remaining instructions. A priori any choice of localization for these instructions will be adequate, since all atomicity constraints have already been satisfied. The problem of localising the remaining sequential instructions is therefore performance related. The main idea is to try to minimize the number of expected communications during the execution of the program, i.e. minimizing the number of thread migrations. For that, a weighted control flow graph [Aho et al., 1998] is constructed for each sequence. Weights in this graph model the expected number of times the control flows through each vertex. The problem then boils down to coloring this graph (each color representing an execution site) while minimizing the weighted number of color changes. This problem has been proven equivalent to the *Multiterminal Cut Problem* [De Wachter et al., 2005; De Wachter, 2005], which is known to be NP-hard [Dahlhaus et al., 1994].

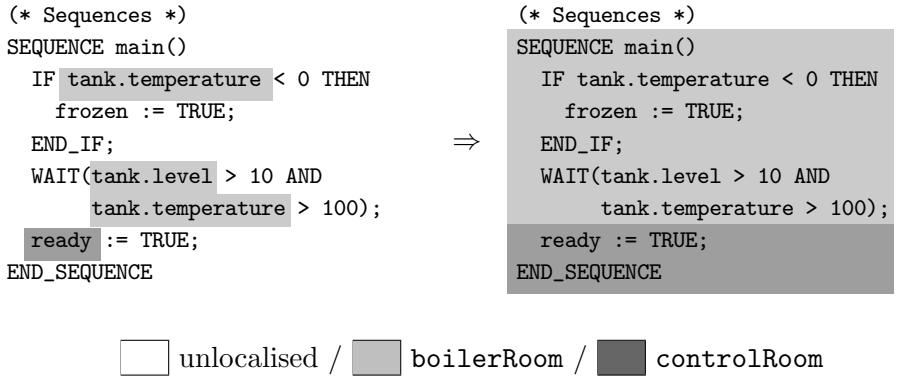


Figure 2.12 - Localising sequential code

Example 2.11

Figure 2.12 illustrates the second step of the distribution process on the program of Figure 2.3. In this particular case, the only instruction that is not already localised after the first step is the instruction at line 58 (`frozen := TRUE`). The instruction appearing right before (`IF` (...) `THEN`) and right after (`WAIT(...)`) are both localised on the site `boilerRoom`. Clearly, in order to minimize the number of thread migration, one must assign this instruction to the `boilerRoom` site.

2.3 Related Works

The problem of distributing applications that control reactive systems has been studied for many years now and several interesting observations made on these studies shaped the design of dSL. We comment in more detail works on process algebras, Synchronous Languages, and higher level frameworks such as Unity.

In the world of process algebras, the problem of automated distribution is defined as a correctness preserving transformation of a *centralized* specification into a semantically equivalent *distributed* one. (e.g. for bisimulation equivalence [Milner, 1989], see [Massart, 1992; Brinksma and Langerak, 1995]). It has also been studied on various types of transition systems [Castellani et al., 1999; Morin, 1999; Stefanescu et al., 1999; Meuter, 2003]. These works solved part of the problem. Indeed, contrary to programming languages, these formalisms do not allow variables, making them impractical.

In a higher level framework, Chandy and Misra have proposed the Unity approach [Chandy and Misra, 1988] to model and design asynchronous or synchronous parallel programs. Let us recall that the principle in Unity, similar to the one proposed by the B method [Abrial, 1996], is to use a design modeling language together with a proof

system to provide, through several design decisions, correct parallel programs. Central in this framework is the separation between the concern of program development and the physical architecture on which it is implemented. The program development phase provides the specification of the Unity program in itself using guarded, multiple assignment statements. During the second phase, starting from a Unity program, a *mapping* to an architecture is used to describe a possible implementation. Various possibilities are proposed to implement a Unity program on a distributed architecture. The program variables can be seen as shared variables or communication can be done through FIFO channels. In both cases, a protocol must be explicitly given to preserve the data integrity or synchronize the execution flow. This implies that, each time the target architecture is modified (e.g. adding or removing processor(s), moving variable(s), refining the distribution, ...), the Unity program has to be modified at the communication level, to fit this new distribution. Therefore, the Unity program must be designed with a desired mapping in mind, which has a negative impact on the reconfiguration flexibility of applications and the transparency of the distribution.

On the programming language side, the most relevant works on automated distribution of reactive systems have been done in the domain of synchronous languages such as Esterel [Berry and Gonthier, 1992], Lustre [Caspi et al., 1987] and Signal [LeGuernic et al., 1991], which answered questions on how to specify controllers in a natural and semantically well defined way. Unfortunately, because the semantics must be preserved, the distribution of synchronous languages suffers from a performance problem which, in practice, may not be acceptable. Indeed, the synchronous programming scheme found in these languages supposes that time is defined as a sequence of *instants* which are common to all parallel processes contained in the specification. Although this allows the use of synchronous broadcast [Berry, 1989; Benveniste and Berry, 1991] resulting in sequential code, a strong synchronization scheme must be used to preserve these instants when such programs are distributed [Girault, 1994; Aubry, 1997]. This strong synchronization has several drawbacks in an industrial environment. First of all, to keep all processes in pace, numerous messages need to be exchanged at each global instant. Secondly, all participating processes have to advance at the speed of the slowest process. Finally, the failure of one of the processes makes the whole system deadlock. To the best of our knowledge, the synchronous approach has no answer to these shortcomings. We believe therefore that, although perfectly suitable for tightly coupled homogeneous systems and having the benefit of simplicity when it comes to specifying a controller, the simplicity of the synchronous approach is too costly in terms of performance when applied to loosely coupled heterogeneous

systems. Moreover, in practice, the strong synchronization of all processes is rarely needed and must therefore not be used as a default.

These observations motivated the design of dSL. At the design level, Unity seems more sound than what is proposed with the direct use of programming languages (among which dSL), since a Unity design goes through the correctness proof for the design before doing the implementation. Unfortunately, the industrial control world has not yet integrated this more formal approach to design real systems, and still uses more specialized programming languages defined as industrial standards. The philosophy of dSL goes therefore in the other direction: it proposes a language close to the used standards. With the drawbacks of synchronous languages in mind, dSL rejects the synchronous product [Milner, 1981] used in synchronous languages and adopts a semantics based on asynchronous composition of local instantaneous code and global distributed code.

Chapter

3

Model Checking dSL Programs

« *All models are wrong, but some are useful* »

George E. P. Box

In the previous chapter, we have introduced the dSL language as a tool to ease the development of distributed systems. The main advantage of dSL is its transparent distribution allowing the programmer to focus on the functional aspects of the systems, while leaving most of the technical details to the compiler. This is only but a first step towards making the development of safety critical applications less error-prone. Indeed, as argued in the introduction, when dealing with critical control applications, thorough validation is needed. One of the most ambitious techniques for that is *model checking*, introduced in the early 1980s by Clarke and Emerson [Clarke and Emerson, 1981]. This technique can be decomposed in three successive steps: modeling, specification and verification.

In this chapter, we study the application of model checking for the verification of dSL programs. First, in Section 3.1, we tackle the modeling by introducing a formal operational semantics for dSL programs. Next, in Section 3.2, we address the second step, namely specification. In particular, we present a remarkable property of the semantics, and show how this property can be exploited to simplify the model checking of LTL specifications. Finally, in Section 3.3, we turn our attention to the last remaining step of model checking process, i.e. the actual verification itself. The content of this chapter is based on a joint work with Bram De Wachter, Alexandre Genon and Thierry Massart, published in [De Wachter et al., 2005], also included in Bram De Wachter's thesis [De Wachter, 2005].

3.1 Modeling

In this section, we explain how a dSL program and its semantics can be formally modeled. In a nutshell, the semantics of a program will be modelled as the parallel composition of k processes (one for each execution site) communicating with one another by message passing. Communications between processes will be point-to-point and modeled using FIFO communication channels, one between each pair of processes. These FIFO communication channels will allow each process to communicate the update of one of its variables to all other processes. This way, each process can maintain its own copy of distant (tilded) variables. Each process will have a cyclic behaviour. Cycles will be divided in three phases. The first one is the *input phase* during which the value of variables linked to sensors are changed according to the physical state of the device they are attached to. During this phase, events depending on those variables may be triggered as well. The second one is the *processing phase* during which (i) incoming messages (for tilded variables) are processed and (ii) sequences are executed. The third and final phase is the *output phase* during which the values of the variables linked to the actuators force the physical state of the devices they are connected to.

Before addressing the heart of the problem, i.e. the semantics itself, we give in Section 3.1.1 and 3.1.2, formal models for programs and distributions. We then proceed to the semantics itself, in Section 3.1.3. Finally, we conclude this section by discussing the history of this semantics.

3.1.1 Modeling a Program

The first step, in order to define a formal semantics of a dSL program, is to give a formal abstract definition of what a program is. For the sake of simplicity, we will focus our attention on the original features of dSL. We will therefore not consider the object-oriented features of the language. This implies that classes and associated methods and class events (`WHEN IN`) will not be considered in this chapter. We will also restrict ourselves to sequences without parameters and to boolean variables. The complete grammar for this subset of dSL can be found in Appendix A.2. Keeping this in mind, we can give an abstract formal definition of a dSL program.

Definition 3.1 (dSL Program)

A dSL program P is a tuple $\langle V, E, S, s_0, \lhd, \Lambda \rangle$, where:

- V is a finite set of boolean variables composed of three disjoint subsets V_{in}, V_{out} and V_τ, denoting respectively input, output and internal variables.
- E is a finite set of events (`WHEN`).
- S is a finite set of sequences.
- $s_0 \in S$ is the initial sequence (the sequence `main`).
- $\lhd \subseteq (V \cup E \cup S) \times (V \cup E \cup S)$ is a total order on the elements of the program representing the order in which those elements are declared.
- $\Lambda \in (V_{in} \cup V_{out}) \mapsto \mathbb{N}$ is the localization function mapping each external variable to an execution site (here modeled as a natural number).

In the following, for an event $e \in E$, we note $\mathsf{cond}(e)$ the condition of e and $\mathsf{body}(e)$, the body of e. For a sequence $s \in S$, we note $\mathsf{body}(s)$ the body of s and $\mathsf{local}(s)$ the set of local variables declared in s. We will also use $\mathsf{var}(\cdot)$, $\widetilde{\mathsf{var}}(\cdot)$ and $\mathsf{instr}(\cdot)$, to denote respectively the set of variables, tilded variables and instructions, appearing in a dSL construct. For blocks, instructions and expressions, $\mathsf{var}(\cdot)$, $\widetilde{\mathsf{var}}(\cdot)$ and $\mathsf{instr}(\cdot)$ is defined inductively on the grammar, as shown in Figure 3.1. Note that for a conditional statement or a loop, the variables appearing in the nested block are not included in $\mathsf{var}(i)$[1]. Moreover, given a sequence $s \in S$, we define $\mathsf{instr}(s) \stackrel{\text{def}}{=} \mathsf{instr}(\mathsf{body}(s))$, $\mathsf{var}(s) \stackrel{\text{def}}{=} \cup_{i \in \mathsf{instr}(s)} \mathsf{var}(i)$ and $\widetilde{\mathsf{var}}(s) \stackrel{\text{def}}{=} \cup_{i \in \mathsf{instr}(s)} \widetilde{\mathsf{var}}(i)$. Similarly, given an event $e \in E$, we define $\mathsf{instr}(e) \stackrel{\text{def}}{=} \mathsf{instr}(\mathsf{body}(e))$, $\mathsf{var}(e) \stackrel{\text{def}}{=} \cup_{i \in \mathsf{instr}(e)} \mathsf{var}(i) \cup \mathsf{var}(\mathsf{cond}(e))$ and $\widetilde{\mathsf{var}}(e) \stackrel{\text{def}}{=} \cup_{i \in \mathsf{instr}(e)} \widetilde{\mathsf{var}}(i) \cup \widetilde{\mathsf{var}}(\mathsf{cond}(e))$. Finally, given a variable $x \in V$, we define the set of events depending on x as $\mathsf{evt}(x) \stackrel{\text{def}}{=} \{e \in E \mid x \in \mathsf{var}(\mathsf{cond}(e))\}$ and the set of events depending on the tilded copy of x as $\widetilde{\mathsf{evt}}(x) \stackrel{\text{def}}{=} \{e \in E \mid x \in \widetilde{\mathsf{var}}(\mathsf{cond}(e))\}$.

In order for the semantics to be defined properly, a dSL program P must satisfy some assumptions. First of all, every variable of the program must be declared and we will assume that at least one variable appears non-tilded in each event. Then, we will also assume that the variables appearing in a sequence are never preceded by the $\sim$ operator. Indeed, as previously mentioned, sequences migrate from sites to sites following the variables that are used. Therefore, using tilded copies of variables instead of the variables themselves does not make much sense. This is therefore prohibited. Finally, we will assume that no local variable declared in a sequence hides any global variable, i.e. they do not have the same name. This leads us to the definition of a *well-formed* dSL program.

[1] This choice was made for technical reasons that will become clear later on

Grammar rule			Attributes		
⟨instruction i⟩	::=	⟨id x⟩ `:=` ⟨rhside e⟩	$\mathsf{instr}(i)$	$\stackrel{\text{def}}{=}$	$\{i\}$
			$\mathsf{var}(i)$	$\stackrel{\text{def}}{=}$	$\{x\} \cup \mathsf{var}(e)$
			$\widetilde{\mathsf{var}}(i)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(e)$
	|	`WAIT` ⟨rhside e⟩	$\mathsf{instr}(i)$	$\stackrel{\text{def}}{=}$	$\{i\}$
			$\mathsf{var}(i)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e)$
			$\widetilde{\mathsf{var}}(i)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(e)$
	|	`LAUNCH` ⟨id s⟩	$\mathsf{instr}(i)$	$\stackrel{\text{def}}{=}$	$\{i\}$
			$\mathsf{var}(i)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\widetilde{\mathsf{var}}(i)$	$\stackrel{\text{def}}{=}$	$\varnothing$
	|	`IF` ⟨rhside e⟩ `THEN` ⟨block b_1⟩ `ELSE` ⟨block b_2⟩ `END_IF`	$\mathsf{instr}(i)$	$\stackrel{\text{def}}{=}$	$\{i\} \cup \mathsf{instr}(b_1) \cup \mathsf{instr}(b_2)$
			$\mathsf{var}(i)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e)$
			$\widetilde{\mathsf{var}}(i)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e)$
	|	`WHILE` ⟨rhside e⟩ `DO` ⟨block b⟩ `END_WHILE`	$\mathsf{instr}(i)$	$\stackrel{\text{def}}{=}$	$\{i\} \cup \mathsf{instr}(b)$
			$\mathsf{var}(i)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e)$
			$\widetilde{\mathsf{var}}(i)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e)$
⟨rhside e⟩	::=	⟨id x⟩	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\{x\}$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
	|	$\sim$ ⟨id x⟩	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\{x\}$
	|	⟨constant⟩	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
	|	`(` ⟨rhside e'⟩ `)`	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e')$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(e')$
	|	⟨rhside e_1⟩ ⟨bin_op⟩ ⟨rhside e_2⟩	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e_1) \cup \mathsf{var}(e_2)$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(e_1) \cup \widetilde{\mathsf{var}}(e_2)$
	|	⟨un_op⟩ ⟨rhside e'⟩	$\mathsf{instr}(e)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(e)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(e')$
			$\widetilde{\mathsf{var}}(e)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(e')$
⟨block b⟩	::=	⟨instruction i⟩ `;` ⟨block b'⟩	$\mathsf{instr}(b)$	$\stackrel{\text{def}}{=}$	$\{i\} \cup \mathsf{instr}(b')$
			$\mathsf{var}(b)$	$\stackrel{\text{def}}{=}$	$\mathsf{var}(i) \cup \mathsf{var}(b')$
			$\widetilde{\mathsf{var}}(b)$	$\stackrel{\text{def}}{=}$	$\widetilde{\mathsf{var}}(i) \cup \widetilde{\mathsf{var}}(b')$
	|	ε	$\mathsf{instr}(b)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\mathsf{var}(b)$	$\stackrel{\text{def}}{=}$	$\varnothing$
			$\widetilde{\mathsf{var}}(b)$	$\stackrel{\text{def}}{=}$	$\varnothing$

Figure 3.1 - Variables and instructions appearing in a dSL construct

Definition 3.2 (Well-formed dSL Program)

A dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ is *well-formed* if and only if:

(i) $\forall e \in E : (\mathsf{var}(e) \cup \widetilde{\mathsf{var}}(e) \subseteq V) \wedge (\mathsf{var}(e) \neq \varnothing)$

(ii) $\forall s \in S : (\widetilde{\mathsf{var}}(s) = \varnothing) \wedge (\mathsf{var}(s) \subseteq V \cup \mathsf{local}(s)) \wedge (\mathsf{local}(s) \cap V = \varnothing)$

In the following, we assume that every dSL program is well-formed.

3.1.2 Modeling a Distribution

The behaviours of a dSL program P will not only depend on P, but also on how it is distributed. Indeed, recall that each global variable has to be assigned to an execution site at compile-time. This assignment will have an influence on the semantics of a program. For external variables, this information is already present in P, in its localization function Λ. For internal variables, however, it is not that simple. In fact there are many different possible assignments, as long as the atomicity constraints imposed by the program are met. That is why we introduce the notion of distribution. Intuitively, a distribution of a program P, is a partition of the set of its variables compatible with the localization of the external variables of this program and respecting its atomicity constraints. Each element in this partition will model one execution site. The first constraint is the atomicity constraint imposed on the event-driven code. If two variables appear (un-tilded) in the same event, they must be localised on the same execution site. The second constraint is the atomicity constraint imposed on each instruction of a sequence. If two global variables appear in the same sequential instruction, they should be localised on the same site[2]. The formal definition follows.

Definition 3.3 (Distribution of a dSL Program)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, a distribution of P is a partition $D \in \Pi(V)$ such that:

(i) $\forall X \in D, \forall x, x' \in X \cap (V_{in} \cup V_{out}) : \Lambda(x) = \Lambda(x')$

(ii) $\forall e \in E, \exists X \in D : \mathsf{var}(e) \subseteq X$

(iii) $\forall s \in S, \forall i \in \mathsf{instr}(s), \exists X \in D : \mathsf{var}(i) \subseteq X \cup \mathsf{local}(s)$

We note $\mathcal{D}_P$ the set of distributions of P.

[2]Remember that $\mathsf{var}(x)$ does not include the variables of the nested blocks.

If a dSL program P cannot be distributed properly, as illustrated on the lamp control program of Figure 2.9 in Chapter 2, the set of possible distributions will simply be empty ($\mathcal{D}_P = \varnothing$). Otherwise, any distribution $D \in \mathcal{D}_P$ naturally induces a partition of E. Indeed, condition (i) of Definition 3.2 ensures that at least one variable appear (non-tilded) in each event, thus implying that each event is univocally localised. This is stated in the following theorem.

Theorem 3.1 (Partition of Events)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and a distribution $D \in \mathcal{D}_P$, let $E_X \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{var}(e) \subseteq X\}$. We have that:

$$\{\ E_X \mid (X \in D) \wedge (E_X \neq \varnothing)\ \} \in \Pi(E)$$

Proof

We proceed by induction on $|E|$.

Initial step If $E = \varnothing$, we have that $\{E_X \mid (X \in D) \wedge (E_X \neq \varnothing)\} = \varnothing \in \Pi(E)$.

Induction step If $E = \{e\} \cup E'$, by induction, we have a partition of E' given by $Q' = \{E'_X \mid (X \in D) \wedge (E'_X \neq \varnothing)\}$. All that needs to be done to construct a partition Q for E is to localise e univocally. For that, we know by Definition 3.2 that $\mathsf{var}(e) \neq \varnothing$, then since $D \in \mathcal{D}_P$, there exists a unique $X \in D$ such that $\mathsf{var}(e) \subseteq X$. If $E'_X = \varnothing$, we take $Q = Q' \cup \{\{e\}\}$. On the other hand if $E'_X \neq \varnothing$, we take $Q = (Q' \setminus \{E'_X\}) \cup \{E'_X \cup \{e\}\}$. In any case, we have that $Q \in \Pi(E)$.

Before moving on to the semantics, let us examine in more detail the structure of $\mathcal{D}_P$. Indeed, for a given program P, we know by Definition 3.3 that $\mathcal{D}_P \subseteq \Pi(V)$. Therefore, the natural order ($\preceq$) on $\Pi(V)$ can be instantiated to $\mathcal{D}_P$.

Definition 3.4 (Coarser/Finer Distribution)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D, D' \in \mathcal{D}_P$. We say that D is a coarser distribution than D' (or equivalently that D' is a finer distribution than D), noted $D \preceq D'$ if and only if $\forall X' \in D', \exists X \in D : X' \subseteq X$.

It is well-known [Priestley and Davey, 2002] that, for a given set X, $\langle \Pi(X), \preceq \rangle$ forms a lattice. In fact, as we will see, this remains true when considering $\langle \mathcal{D}_P, \preceq \rangle$. However, in order to prove this result, we need one intermediate result.

Lemma 3.1

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D, D' \in \mathcal{D}_P$, we have that:

$$\{X \cap X' \mid (X \in D) \wedge (X' \in D') \wedge (X \cap X' \neq \varnothing)\} \in \mathcal{D}_P$$

Proof

Let $D'' = \{X \cap X' \mid (X \in D) \wedge (X' \in D') \wedge (X \cap X' \neq \varnothing)\}$. We have to prove that $\forall X'' \in D''$, the three conditions of Definition 3.3 of distribution are met.

(*i*) By Definition 3.3, we have that $\forall X \in D, \forall x, x' \in X \cap (V_{in} \cup V_{out}) : \Lambda(x) = \Lambda(x')$ and that $\forall X' \in D', \forall x, x' \in X' \cap (V_{in} \cup V_{out}) : \Lambda(x) = \Lambda(x')$ which directly implies $\forall X'' \in D'', \forall x, x' \in X'' \cap (V_{in} \cup V_{out}) : \Lambda(x) = \Lambda(x')$.

(*ii*) By Definition 3.3, we have that $\forall e \in E : (\exists X \in D : \mathsf{var}(e) \subseteq X) \wedge (\exists X' \in D : \mathsf{var}(e) \subseteq X')$ which implies that $\forall e \in E, \exists X \in D, \exists X' \in D' : \mathsf{var}(e) \subseteq X \cap X'$. But, we know by Definition 3.2 of program that $\mathsf{var}(e) \neq \varnothing$, hence that $X \cap X' \neq \varnothing \in D''$. We can therefore conclude that $X \cap X' \neq \varnothing \in D''$ which implies that $\forall e \in E, \exists X'' \in D'' : \mathsf{var}(e) \subseteq X''$.

(*iii*) By Definition 3.3, we have that $\forall s \in S, \forall i \in \mathsf{instr}(s) : (\exists X \in D : \mathsf{var}(i) \subseteq X \cup \mathsf{local}(s)) \wedge (\exists X' \in D' : \mathsf{var}(i) \subseteq X' \cup \mathsf{local}(s))$ which implies that $\forall s \in S, \forall i \in \mathsf{instr}(s), \exists X \in D, \exists X' \in D' : \mathsf{var}(i) \subseteq (X \cap X') \cup \mathsf{local}(s)$. Then, either $\mathsf{var}(i) \not\subseteq \mathsf{local}(s) \neq \varnothing$, in which case $X \cap X' \neq \varnothing$ and $\forall s \in S, \forall i \in \mathsf{instr}(s), \exists X'' \in D'' : \mathsf{var}(i) \subseteq X'' \cup \mathsf{local}(s)$, or, $\mathsf{var}(i) \subseteq \mathsf{local}(s)$, in which case $\forall X'' \in D'' : \mathsf{var}(i) \subseteq X'' \cup \mathsf{local}(s)$ (which is stronger).

We can now prove that $\langle \mathcal{D}_P, \preceq \rangle$ forms a lattice.

Theorem 3.2 (Lattice of Distributions)

Given a dSL program P, the tuple $\langle \mathcal{D}_P, \preceq \rangle$ forms a lattice.

Proof

First, we assume without loss of generality that $\mathcal{D}_P \neq \varnothing$. Indeed, if $\mathcal{D}_P = \varnothing$, the result is immediate. The remainder of the proof is in two parts. We first prove the existence of the least upper bound. Then, we prove the existence of the greatest lower bound.

(*i*) Let $U = \{X \cap X' \mid (X \in D) \wedge (X' \in D') \wedge (X \cap X' \neq \varnothing)\}$. We prove that, in fact, $U = \mathsf{lub}_{\preceq}(\{D, D'\})$. First, by Lemma 3.1, we know that $U \in \mathcal{D}_P$. Then, by construction, we have that $D \preceq U$ and $D' \preceq U$. Finally, $\forall U' \in \mathcal{D}_P$ such that $D \preceq U'$ and $D' \preceq U'$, by Definition 3.4 of finer/coarser distribution, we have that $\forall Y' \in U' : (\exists X \in D : Y' \subseteq X) \wedge (\exists X' \in D' : Y' \subseteq X')$, which implies that $\forall Y' \in U', \exists X \in D, \exists X' \in D' : Y' \subseteq X \cap X'$. Then, since $Y' \neq \varnothing$, we have that $\forall Y' \in U', \exists Y \in U : Y' \subseteq Y$ which, in turn implies that $U \preceq U'$. We can therefore conclude that $U = \mathsf{lub}_{\preceq}(\{D, D'\})$.

```
1 VAR
2     x,y,z,t,u,b: INT;
3 END_VAR
4
5 WHEN x > 0 THEN
6     y = y+1;
7 END_WHEN
```

```
 9 SEQUENCE main()
10     z := 1;
11     t := 10;
12     u := 2 * t;
13     v := 0;
14 END_SEQUENCE
```

Figure 3.2 - A bogus dSL program

Proof (cont'd)

(*ii*) We prove equivalently that the set $\mathsf{LB}_{\preceq}(\{D, D'\})$ of lower bounds of D and D' has a unique maximal element. First, we know that $\mathsf{LB}_{\preceq}(\{D, D'\}) \neq \varnothing$, because since $\mathcal{D}_P \neq \varnothing$, we have that $\{V\} \in \mathsf{LB}_{\preceq}(\{D, D'\})$. Therefore, the only other possibility is that the set contains two maximal elements. We show that this is impossible. Indeed, assume the existence of two different incomparable maximal elements $L, L' \in \mathsf{LB}_{\preceq}(\{D, D'\})$. Let us construct $L'' = \{X \cap X' \mid (X \in L) \wedge (X' \in L') \wedge (L \cap L' \neq \varnothing)\}$. By Lemma 3.1, we know that $L'' \in \mathcal{D}_P$, and by construction, we have that $L \preceq L''$ and $L' \preceq L''$. Moreover, we know that $L \preceq D$ and $L \preceq D'$ which implies, by Definition 3.3 of distribution, that $\forall Y \in D : (\exists X \in L : Y \subseteq X) \wedge (\exists X' \in L' : Y \subseteq X')$. It follows that $\forall Y \in D, \exists X \in L, \exists X' \in L' : Y \subseteq X \cap X'$, and since $Y \neq \varnothing$, that $\forall Y \in D, \exists X'' \in L'' : Y \subseteq X''$. We can therefore conclude that $L'' \preceq D$. A similar argument can be used to prove that $L'' \preceq D'$. Therefore, we can conclude that $L'' \in \mathsf{LB}_{\preceq}(\{D, D'\})$ which contradicts the fact that L and L' are maximal in $\mathsf{LB}_{\preceq}(\{D, D'\})$.

It follows, in particular, that since $\mathcal{D}_P$ is finite, there exists a unique *minimal* distribution, defined as $D_{\mathsf{min}} \stackrel{\text{def}}{=} \mathsf{glb}_{\preceq}(\mathcal{D}_P) = \{V\}$ and a unique *maximal* distribution, defined as $D_{\mathsf{max}} \stackrel{\text{def}}{=} \mathsf{lub}_{\preceq}(\mathcal{D}_P)$. Furthermore, since $\mathcal{D}_P$ is finite, the lattice $\langle \mathcal{D}_P, \preceq \rangle$ is complete. However, as illustrated in the following example, is not distributive.

Example 3.1 (Example of Program and Associated Lattice of Distributions)

Figures 3.2 and 3.3 show respectively an example of dSL program and its associated lattice of distribution. This lattice is not distributive. Indeed, consider the four distributions $D_1 \stackrel{\text{def}}{=} \{\{x, y, v\}, \{z, t, u\}\}$, $D_2 \stackrel{\text{def}}{=} \{\{x, y, z, t, u\}, \{v\}\}$, $D_3 \stackrel{\text{def}}{=} \{\{x, y\}, \{z, t, u, v\}\}$ and $D_4 = \{\{x, y\}, \{z, t, u\}, \{v\}\}$. We have that $D_1 \sqcap (D_2 \sqcup D_3) = D_1 \sqcap D_4 = D_1 \neq (D_1 \sqcap D_2) \sqcup (D_1 \sqcap D_3) = D_4 \sqcup D_4 = D_4$.

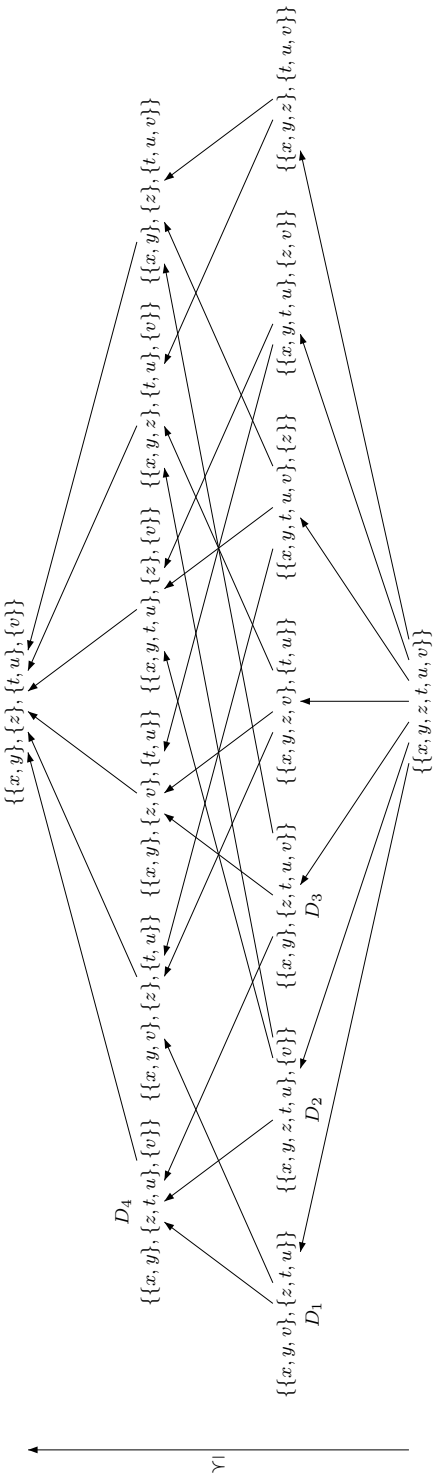


Figure 3.3 - The lattice of distributions for the program of Figure 3.2

3.1.3 Semantics of a Program

Now that we have all the necessary tools, we can start to give a formalization of the semantics of a dSL program. This semantics will be given in terms of Kripke structures. The first step is therefore is to describe the states of that Kripke structure. First of all, each state in the semantics includes the local state of each process. In fact, the local state of a process i contains three components:

(i) a block of instructions remaining to be executed, called the *workload*. On top of regular dSL instructions, this workload contains some new instructions that are used for internal treatments. First, `INPUT`(x), resp. `OUTPUT`(x), is used for the input, resp. output, of a variable x. Similarly, `BCAST`(x,b) denotes the broadcast of a variable x and its boolean value b. Finally, `MSG`, resp. `SEQ`, is used to indicate that the process should treat his messages, resp. sequences.

(ii) a *valuation* for all variables needed by process i. These variables include the variables controlled by process i, the asynchronous (tilded) copy of distant variables, including its own variables. For a variables $x \in V$, we note $\tilde{x}$ its tilded copy. A shadow copy of each output variables governed by process i is also needed. These variables are used instead of the actual output variables during the processing phase, since the real output variables are only updated during the output phase. For an output variables $x \in V_{out}$, we note $\hat{x}$ its shadow copy. Finally, some variables c_e are needed to store the result of the last known evaluation of the condition of each event e. These variables are used to detect rising-edges in the condition of events. For an event $e \in E$, we note c_e its corresponding variable.

(iii) the content of all receiving communication channels. Each message in these channel is a pair $\langle x, v \rangle$ where x is a variable and v its value. Moreover, $\diamond$ markers are used to delimit the messages left to be treated in the current cycle of process i.

The definition of a local state of a process follows.

Definition 3.5 (Local State of a Process)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ and a distribution $D = \{X_1, ..., X_k\}$ of P, a local state of process $i \in [1, k]$ is a tuple $\langle \omega_i, \nu_i, \phi_i \rangle$ where:

- ω_i is the workload of process i,
- $\nu_i \in (X_i \cup \{\tilde{x} \mid x \in V\} \cup \{\hat{x} \mid x \in X_i \cap V_{out}\} \cup \{c_e \mid e \in E_{X_i}\}) \mapsto \mathbb{B}$ is the local valuation of process i,
- $\phi_i \in [1, k] \mapsto ((V \times \mathbb{B}) \cup \{\diamond\})^*$ is the valuation for the FIFO channels.

Grammar rule	Attributes
$\langle \text{rhside } e \rangle ::= \langle \text{id } x \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \begin{cases} v(\hat{x}) & \text{if } x \in V_{out} \\ v(x) & \text{otherwise} \end{cases}$
$\mid \sim \langle \text{id } x \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} v(\tilde{x})$
$\mid$ `TRUE`	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{tt}$
$\mid$ `FALSE`	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{ff}$
$\mid$ `(` $\langle \text{rhside } e' \rangle$ `)`	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{eval}[v](e')$
$\mid \langle \text{rhside } e_1 \rangle$ `AND` $\langle \text{rhside } e_2 \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{eval}[v](e_1) \wedge \mathsf{eval}[v](e_2)$
$\mid \langle \text{rhside } e_1 \rangle$ `OR` $\langle \text{rhside } e_2 \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{eval}[v](e_1) \vee \mathsf{eval}[v](e_2)$
$\mid \langle \text{rhside } e_1 \rangle$ `<>` $\langle \text{rhside } e_2 \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \neg(\mathsf{eval}[v](e_1) \Leftrightarrow \mathsf{eval}[v](e_2))$
$\mid \langle \text{rhside } e_1 \rangle$ `==` $\langle \text{rhside } e_2 \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \mathsf{eval}[v](e_1) \Leftrightarrow \mathsf{eval}[v](e_2)$
$\mid$ `NOT` $\langle \text{rhside } e' \rangle$	$\mathsf{eval}[v](e) \stackrel{\text{def}}{=} \neg\mathsf{eval}[v](e')$

Figure 3.4 - Valuations extended to dSL expressions

In the previous definition, $\phi_i(j)$ denotes the messages sent by process j to process i. For the sake of readability, it is noted $\phi_{i \leftarrow j}$ in the following.

Moreover, sequences are executed by the local processes during their *processing phase*. The state of these sequences also needs to be included in the global states of the semantics. Let us define the local state of a sequence.

Definition 3.6 (Local State of a Sequence)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, a local state of a sequence $s_j \in S$ is a tuple $\langle \sigma_j, \mu_j \rangle$ where:

- σ_j is the block of code left to be executed in s_j.
- $\mu_j \in \mathsf{local}(s_j) \mapsto \mathbb{B}$ is the valuation for the local variables of s_j

Using the local valuation of the processes and the valuations for the local variables of the sequences, dSL expressions can be evaluated. Given a valuation v, and a dSL expression e, we note $\mathsf{eval}[v](e)$ the evaluation of e using the valuation v. Given a valuation v, $\mathsf{eval}[v]$ is defined inductively on the grammar, as shown in Figure 3.4.

Now that we have all the building blocks, we can define a global state of a dSL program. Such a global state is composed of the local state of each process and the local state of each sequence. This can be formalized as follows.

Definition 3.7 (Global State of a dSL Program)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ with a set of sequences $S = \{s_1, s_2, ..., s_n\}$ and a distribution $D = \{X_1, X_2, ..., X_k\}$ of P, a global state of P w.r.t. D is a tuple $q = \langle \ell_1, \ell_2, ...\ell_k, m_0, m_1, ..., m_n \rangle$ where:

- $\forall i \in [1, k] : \ell_i = \langle \omega_i, \nu_i, \phi_i \rangle$ is a local state of process i,
- $\forall j \in [1, n] : m_j = \langle \sigma_j, \mu_j \rangle$ is a local state of sequence s_j

We now focus our attention to the transition relation of the KS. This transition relation will be defined operationally, i.e. using operational semantics rules. In order to simplify the presentation of those rules, we need some additional notations. First, we need a shortcut for the treatment of events. Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and a subset $X \subseteq E$, we define the treatment of the events in X, noted $\mathsf{treat}(X)$, inductively as follows:

$$\mathsf{treat}(X) \stackrel{\text{def}}{=} \begin{cases} \left[\begin{array}{l} \texttt{IF } \mathsf{cond}(e) \texttt{ AND NOT } c_e \texttt{ THEN} \\ \quad c_e{:=}\texttt{TRUE}; \mathsf{body}(e) \\ \texttt{ELSE} \\ \quad c_e{:=}\mathsf{cond}(e); \\ \texttt{END\_IF}; \mathsf{treat}(X \setminus \{e\})) \end{array}\right] & \text{if } X \neq \varnothing \wedge e = \mathsf{min}_{\lhd}(X) \\ \varepsilon & \text{if } X = \varnothing \end{cases}$$

We also need a shortcut to describe the input and output phases. Given a dSL program $P = \langle V, E, S, \lhd, \Lambda \rangle$, and a subset $X \subseteq V$, we define the sampling of X, respectively update of X, noted $\mathsf{in}(X)$, respectively $\mathsf{out}(X)$, inductively as follows:

$$\mathsf{in}(X) \stackrel{\text{def}}{=} \begin{cases} \texttt{INPUT}(x); \mathsf{in}(X \setminus \{x\}) & \text{if } X \neq \varnothing \wedge x = \mathsf{min}_{\lhd}(X) \\ \varepsilon & \text{if } X = \varnothing \end{cases}$$

$$\mathsf{out}(X) \stackrel{\text{def}}{=} \begin{cases} \texttt{OUTPUT}(x); \mathsf{out}(X \setminus \{x\}) & \text{if } X \neq \varnothing \wedge x = \mathsf{min}_{\lhd}(X) \\ \varepsilon & \text{if } X = \varnothing \end{cases}$$

Finally, for all $X_i \in D$, we note $E_{X_i} \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{var}(e) \subseteq X_i\}$ the set of events governed by process i. We can now proceed with the presentation of the semantic rules. In these rules, we assume the existence of a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ and a distribution $D = \{X_1, ..., X_k\}$ of P. Each rule is of the form

$$\frac{cond}{\langle P, D \rangle \vdash q \hookrightarrow q'}$$

where *cond* is the condition for the rule to be applied, and where q, respectively q', is the global state of P before, respectively after, the rule is applied.

Cycle Start Rule Rule (3.1) corresponds to the beginning of a new cycle. If, at a point in the execution, the workload of process i becomes empty (ε), then a new cycle is scheduled in its workload. As already mentioned, each cycle is divided in three phases. The first phase is the *input phase*, where all input variables governed by process i are sampled and the corresponding events are treated. The second phase is the *processing phase*, where messages from other processes are treated and sequences are executed. The third and last phase is the *output phase*, where all output variables governed by process i are written. Moreover, $\diamond$ markers are inserted, non-deterministically, in the receiving communication channels of process i. In each communication channel, messages appearing before the first $\diamond$ marker are the messages that must be treated during this cycle. This will prevent process i to read messages that are received during this cycle.

$$\frac{(\omega_i = \varepsilon)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \mathsf{in}(X_i \cap V_{in}); \mathsf{treat}(E_{X_i}); \mathtt{MSG}; \mathtt{SEQ}; \mathsf{out}(X_i \cap V_{out}), \nu_i, \phi_i'\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.1}$$

where $\forall j \in [1, k] : \phi'_{i \leftarrow j} \in (\phi_{i \leftarrow j} \parallel \diamond)$.

Input Rules Rule (3.2) describes the sampling of one input variable. In this case, the valuation of the input variable is updated accordingly, and the new value is scheduled to be broadcast to all other processes.

$$\frac{(\omega_i = \mathtt{INPUT}(x); \omega_i') \wedge (a \in \mathbb{B})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \mathtt{BCAST}(x, a); \omega_i', \nu_i[x \mapsto a], \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.2}$$

Broadcast Rule Rule (3.3) describes the broadcast of the value of one variable to all processes. If a process has to broadcast the value of certain variable, then a transition is taken, leading to a global state where all the receiving communication channels are updated with a message containing the variable and its value. Note that a message is also appended to the channel of the process performing the broadcast. Indeed, this local process might use the tilded copy of its own variable, and this copy needs to be (asynchronously) updated as well.

$$\frac{(\omega_i = \mathtt{BCAST}(x, v); \omega_i')}{\begin{array}{c}\langle P, D\rangle \vdash \langle \langle \omega_1, \nu_1, \phi_1\rangle, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \langle \omega_k, \nu_k, \phi_k\rangle, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \langle \omega_1, \nu_1, \phi_1'\rangle, ..., \langle \omega_i', \nu_i, \phi_i'\rangle, ..., \langle \omega_k, \nu_k, \phi_k'\rangle, m_0, ..., m_n\rangle\end{array}} \tag{3.3}$$

where $\forall j \in [1, k] : \phi'_j = \phi_j[i \mapsto \phi_{j \leftarrow i} \cdot \langle x, v\rangle]$.

Message Treatment Rules Rule (3.4) describes the treatment of a message during the processing phase. If there is some message left to be treated at the beginning of one of the receiving communications channels in process i, this message must be treated. The valuation in process i is updated, the events that need to be treated are scheduled in the workload and finally, the message is removed from the channel.

$$\frac{(\omega_i = \texttt{MSG}; \omega_i') \wedge (\exists j \in [1,k] : \phi_{i \leftarrow j} = \langle x, v \rangle \cdot \psi)}{\begin{array}{c}\langle P, D \rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap E_{X_i}); \omega_i, \nu_i[\tilde{x} \mapsto v], \phi_i[j \mapsto \psi] \rangle, ..., \ell_k, m_0, ..., m_n \rangle\end{array}} \tag{3.4}$$

End of Message Treatment Rule Rule (3.5) describes the end of message treatment during the processing phase. If all messages to be treated in this cycle have been treated, i.e. the first element in each receiving channel is a $\diamond$ marker, the markers are simply removed from each channels, and the `MSG` instruction is removed from the workload.

$$\frac{(\omega_i = \texttt{MSG}; \omega_i') \wedge (\forall j \in [1,k] : \phi_{i \leftarrow j} = \diamond \cdot \phi'_{i \leftarrow j})}{\langle P, D \rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle \hookrightarrow \langle \ell_1, ..., \langle \omega_i', \nu_i, \phi_i' \rangle, ..., \ell_k, m_0, ..., m_n \rangle} \tag{3.5}$$

Output Rule Rule (3.6) describes the update of an output variable. In this case, the value of the corresponding shadow copy is used to update the actual output variable.

$$\frac{(\omega_i = \texttt{OUTPUT}(x); \omega_i')}{\begin{array}{c}\langle P, D \rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i', \nu_i[x \mapsto \nu_i(\hat{x})], \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle\end{array}} \tag{3.6}$$

Assignment Rules Rules (3.7) to (3.9) describe the treatment of an assignment in the workload. Three cases need to be considered. First, if the assigned variable corresponds to a variable c_e for some event e, i.e. the last evaluation of the condition of an event e, the valuation is simply updated.

$$\frac{(\omega_i = x := v; \omega_i') \wedge (x \in \{c_e \mid e \in E\})}{\begin{array}{c}\langle P, D \rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i', \nu_i[x \mapsto \mathsf{eval}[\nu_i](v)], \phi_i \rangle, ..., \ell_k, m_0, ..., m_n \rangle\end{array}} \tag{3.7}$$

The second case is when the assigned variables is an output variable, in which case, the valuation of its corresponding shadow copy is updated, the new value is scheduled to be broadcast and all events depending on this variable are scheduled for treatment.

$$\frac{(\omega_i = x := v; \omega_i') \wedge (x \in X_i \cap V_{out})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \texttt{BCAST}(x, \mathsf{eval}[\nu_i](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega_i', \nu_i[\hat{x} \mapsto \mathsf{eval}[\nu_i](v)], \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.8}$$

The remaining case is when the assigned variable is not an output variable, in which case, the valuation is simply updated, the new value is broadcast and all events depending on that variable are scheduled for treatment.

$$\frac{(\omega_i = x := v; \omega_i') \wedge (x \in X_i \setminus V_{out})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \texttt{BCAST}(x, \mathsf{eval}[\nu_i](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega_i', \nu_i[x \mapsto \mathsf{eval}[\nu_i](v)], \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.9}$$

If Rules Rules (3.10) and (3.11) correspond to the treatment of a conditional statement. As expected, if the condition evaluates to tt, the code in the first branch is inserted in the workload.

$$\frac{(\omega_i = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF}; \omega_i') \wedge (\mathsf{eval}[\nu_i](e) = \mathsf{tt})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_{\texttt{THEN}}; \omega_i', \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.10}$$

On the other hand, if the condition evaluates to ff, the code of the second branch is inserted.

$$\frac{(\omega_i = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF}; \omega_i') \wedge (\mathsf{eval}[\nu_i](e) = \mathsf{ff})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_{\texttt{ELSE}}; \omega_i', \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.11}$$

Launch Rules Rules (3.12) and (3.13) describe the treatment of a `LAUNCH` instruction. If some process has a `LAUNCH` instruction at the beginning of its workload, and if the sequence is not already running (i.e the workload in the local state of the corresponding

sequence is empty), then the body of that sequence is scheduled for execution and all the local variables of that sequences are reset to ff

$$\frac{(\omega_i = \texttt{LAUNCH}\ s_j; \omega_i') \wedge (\sigma_j = \varepsilon)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i', \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \mathsf{body}(s_j), \mu_j[\mathsf{local}(s_j) \mapsto \mathsf{ff}]\rangle, ..., m_n\rangle\end{array}} \tag{3.12}$$

On the other hand, if the sequence in question is already running, the `LAUNCH` is simply discarded.

$$\frac{(\omega_i = \texttt{LAUNCH}\ s_j; \omega_i') \wedge (\sigma_j \neq \varepsilon)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i', \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.13}$$

Sequence Assignment Rules Rules (3.14) and (3.15) describe how an assignment is handled in a sequence. A sequence instruction will only be executed by a process having a `SEQ` internal instruction at the beginning of its workload. This is also true in all other rules corresponding to the treatment of an instruction inside a sequence. For the assignment, there are two cases to consider. The first case is when the assigned variables is local to the sequence, in which case, its local valuation needs to be updated. For that, we need to determine by which process the right-hand side of the assignment can be evaluated. Then, as expected, the valuation of the variable is updated with its new value.

$$\frac{(\sigma_j = x := e; \sigma_j') \wedge (\omega_i = \texttt{SEQ}; \omega_i') \wedge (x \in \mathsf{local}(s_j)) \wedge (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j))}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j', \mu_j[x \mapsto \mathsf{eval}[\mu_j \cup \nu_i](e)]\rangle, ..., m_n\rangle\end{array}} \tag{3.14}$$

The second case is when the assigned variable belongs to process i. In this case, the assignment must be executed by this process. Therefore, after evaluating the right hand side expression, the assignment is transferred to its workload.

$$\frac{(\sigma_j = x := e; \sigma_j') \wedge (\omega_i = \texttt{SEQ}; \omega_i') \wedge (x \in X_i)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle x := \mathsf{eval}[\mu_j \cup \nu_i](e); \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j', \mu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.15}$$

Sequence If Rules Rules (3.16) and (3.17) describe how a conditional statement is handled in a sequence. The first thing to do is to determine by which process the condition can be evaluated and then to determine its value. For that, one must use both the valuation of that process and the valuation of the sequence. Then, as expected, if the condition is evaluated to tt, the first branch is inserted in the woarkload of the sequence.

$$\frac{\begin{array}{c}(\sigma_j = \texttt{IF}\ e\ \texttt{THEN}\ \sigma_{\texttt{THEN}}\ \texttt{ELSE}\ \sigma_{\texttt{ELSE}}\ \texttt{END\_IF}; \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i) \wedge \\ (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j)) \wedge (\mathsf{eval}[\mu_j \cup \nu_i](e) = \mathsf{tt})\end{array}}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_{\texttt{THEN}}; \sigma'_j, \nu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.16}$$

On the other hand, if the condition evaluates to ff, the code of the second branch is inserted.

$$\frac{\begin{array}{c}(\sigma_j = \texttt{IF}\ e\ \texttt{THEN}\ \sigma_{\texttt{THEN}}\ \texttt{ELSE}\ \sigma_{\texttt{ELSE}}\ \texttt{END\_IF}; \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i) \wedge \\ (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j)) \wedge (\mathsf{eval}[\mu_j \cup \nu_i](e) = \mathsf{ff})\end{array}}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_{\texttt{ELSE}}; \sigma'_j, \nu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.17}$$

Sequence Launch Rule Rule (3.18) describes the treatment of a `LAUNCH` instruction inside a sequence. This instruction is simply moved to the workload of any process in its processing phase. Indeed, this instruction does not handle any variable. Therefore, no restriction is imposed on where it should be executed.

$$\frac{(\sigma_j = \texttt{LAUNCH}\ s_h; \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \texttt{LAUNCH}\ s_h; \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma'_j, \mu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.18}$$

Sequence While Rules Rules (3.19) and (3.20) describe how loops are handled inside a sequence. Similarly to what is done in Rules (3.16) and (3.17), the first thing to do is to determine where the condition of the loop can be evaluated. Then we need to consider two cases, depending on this value. If the condition is evaluated to tt, the

loop is scheduled for another turn.

$$\frac{\begin{array}{c}(\sigma_j = \texttt{WHILE}\ e\ \texttt{DO}\ \sigma_b\ \texttt{END\_WHILE}; \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i) \wedge \\ (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j)) \wedge (\mathsf{eval}[\mu_j \cup \nu_i](e) = \mathsf{tt})\end{array}}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_b; \sigma_j, \mu_j\rangle ..., m_n\rangle\end{array}} \tag{3.19}$$

On the other hand, if the condition is evaluated to ff, the loop is terminated.

$$\frac{\begin{array}{c}(\sigma_j = \texttt{WHILE}\ e\ \texttt{DO}\ \sigma_b\ \texttt{END\_WHILE}; \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i) \wedge \\ (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j)) \wedge (\mathsf{eval}[\mu_j \cup \nu_i](e) = \mathsf{ff})\end{array}}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma'_j, \mu_j\rangle ..., m_n\rangle\end{array}} \tag{3.20}$$

Sequence Wait Rule Rule (3.21) describes the treatment of a `WAIT` instruction. The `WAIT` instruction blocks the execution of the sequence until a certain condition is fulfilled. Again, similarly to what is done in Rules (3.16) to (3.20), the first thing to do is to determine where the condition can be evaluated. The `WAIT` is executed only if its condition is evaluated to tt.

$$\frac{(\sigma_j = \texttt{WAIT}(e); \sigma'_j) \wedge (\omega_i = \texttt{SEQ}; \omega'_i) \wedge (\mathsf{var}(e) \subseteq X_i \cup \mathsf{local}(s_j)) \wedge (\mathsf{eval}[\mu_j \cup \nu_i](e)) = \mathsf{tt})}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma_j, \mu_j\rangle ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., \langle \sigma'_j, \mu_j\rangle, ..., m_n\rangle\end{array}} \tag{3.21}$$

End of Sequence Treatment Rule Finally, Rule 3.22 describes the end of sequence treatment. Non deterministically, the treatment of sequences can be stopped at any time, thus ending the processing phase. This models the fact that the execution of a sequence is not synchronized, with the cycles of the processes of the system.

$$\frac{\omega_i = \texttt{SEQ}; \omega'_i}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \omega'_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_0, ..., m_n\rangle\end{array}} \tag{3.22}$$

Now that we have defined all the pieces of the puzzle, we can at last give the formal definition of the semantics of a dSL program P w.r.t. a certain distribution D of P. The only thing left to formalize is the set of initial states and the labelling function. This is taken care of in the following definition.

Definition 3.8 (Semantics of a dSL Program)

Given a dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ with $S = \{s_0, s_1, s_2, ..., s_n\}$, and a distribution $D = \{X_1, X_2, \ldots, X_k\}$ of P, the semantics of P w.r.t. D is a Kripke structure $K_{P,D} = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ over the set of proposition V, where:

- Q is the set of global states of P w.r.t D.
- $I \stackrel{\text{def}}{=} \{\langle \langle \omega_1, \nu_1, \phi_1 \rangle, \ldots \langle \omega_k, \nu_k, \phi_k \rangle, \langle \sigma_0, \mu_0, \rangle, \ldots, \langle \sigma_n, \mu_n \rangle \rangle\}$ where
  - (i) $\forall i \in [1,k] : (\omega_i \stackrel{\text{def}}{=} \varepsilon) \wedge (\nu_i \stackrel{\text{def}}{=} \lambda x \cdot \mathsf{ff}) \wedge (\phi_i \stackrel{\text{def}}{=} \lambda j \cdot \varepsilon)$
  - (ii) $\forall j \in [1,n] : (\sigma_j \stackrel{\text{def}}{=} \varepsilon) \wedge (\mu_j \stackrel{\text{def}}{=} \lambda x \cdot \mathsf{ff})$
  - (iii) $(\sigma_0 = \texttt{BODY}(\mathtt{s}_0)) \wedge (\mu_0 \stackrel{\text{def}}{=} \lambda x \cdot \mathsf{ff})$
- $\mathcal{L} \stackrel{\text{def}}{=} \lambda q \cdot \left(\bigcup_{i \in [1,k]} \{x \in X_i \mid \nu_i(x) = \mathsf{tt}\} \right)$
- $\rightarrow \stackrel{\text{def}}{=} \{\langle q, q' \rangle \in Q \times Q \mid \langle P, D \rangle \vdash q \hookrightarrow q'\}$

In other words, the semantics of a dSL program P w.r.t. a distribution D of P is a Kripke structure, where states are global states of P and transitions between states are defined using the semantics rules presented previously. The only initial state is a state where (i) each processes has an empty workload, all of its local variables set to false, and all incoming communications channels empty; and (ii) each sequence has an empty workload and all its local variables set to false, except (iii) the initial, i.e. `main`, sequence which body is prepared for execution. Finally, each state in this semantics is labeled by the set of variables that are true in this state.

3.1.4 History

The semantics of dSL has greatly evolved since it was first introduced in a joint work with Bram De Wachter, Alexandre Genon and Thierry Massart [De Wachter et al., 2005]. At the time, it did not include sequences yet. It was revisited and extended in [De Wachter, 2005, sec. 3.4] to include them explicitly. Furthermore, the way communications were handled was slightly modified. In [De Wachter et al., 2005], each process was equipped with *one* FIFO communication channel, in which each message destined to that process was added. This assumption was quite restrictive because it imposed two messages from different sources to be received in the order they were sent. To lift this restriction, one communication channel should be used between each pair of processes. The author of [De Wachter, 2005] chose to simulate this by encoding the content of the k "virtual" receiving FIFO channels of one process into *one* cleverly manipulated channel structure. The semantics presented in this document has again

undergone some modifications since then. First, in both [De Wachter et al., 2005] and [De Wachter, 2005], the semantics was presented in terms of a labelled transition system. Unfortunately, most model checking tools, are state-based, i.e. properties are defined on states rather than transitions. That is why, we chose here to define the semantics in terms of a Kripke structure instead. This motivates the introduction of the shadow copy for the output variables. Furthermore, in the semantics presented in [De Wachter et al., 2005] and [De Wachter, 2005], the execution of a sequence and the reception and treatment of a message were allowed to interleave. Those interleavings greatly increased the number of states in the semantics, and are not necessary since, in the actual implementation, the incoming messages are treated at the beginning of the processing phase. We therefore modified the semantics by introducing the `SEQ` internal instruction in order to prevent that. Finally, for sake of clarity, we also introduced some changes in the presentation. For instance, we chose to make the communication channels between each pair of processes explicit. We also integrated the `WAIT` instructions into the semantics instead of removing them syntactically beforehand.

3.2 Specification

Now that the modeling step if taken care of, the next step is to address specification. Defining the semantics in terms of Kripke structures, as we have done in the previous section, allows us to use traditional temporal logics like LTL or CTL. In this particular context, formulae are defined over the set of variables of the program. For instance, the fact that an alarm in a control system is never triggered can simply be specified in LTL as $\neg(\mathsf{F}\,\texttt{alarm})$, or in CTL as $\neg(\mathsf{EF}\,\texttt{alarm})$. In practice, of course, dSL programs will contain integer variables, in which case, propositions will be constraints on those variables. For instance, the fact that the temperature never exceeds a certain limit could be expressed in LTL as $\mathsf{G}(\texttt{temperature} < \texttt{LIMIT})$, or in CTL as $\mathsf{AG}(\texttt{temperature} < \texttt{LIMIT})$.

Unfortunately, as we have seen in Section 3.1, the semantics of a dSL program P is parametrized by one of its distributions. If we want to prove that P is correct independently of any distribution, we must therefore show that the specification is satisfied for *all* distribution $D \in \mathcal{D}_P$. Formally, we say that a well-formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$ satisfies a LTL formula φ, noted $P \models_{\mathsf{L}} \varphi$, if and only if $\forall D \in \mathcal{D}_P : K_{D,P} \models_{\mathsf{L}} \varphi$. Similarly, a program P satisfies a CTL formula φ if and only if $\forall D \in \mathcal{D}_P : K_{P,D} \models_{\mathsf{C}} \varphi$. Given a well-formed dSL program P and LTL formula φ, the LTL model checking problem for dSL (dSL-LTL-MC) consists in determining if P satisfies φ. The CTL model checking problem for dSL (dSL-CTL-MC) is defined similarly.

Model checking a dSL program therefore requires to examine a possibly exponential number of distributions, which can be expensive in general. Fortunately, in the case of LTL, or more precisely LTL-X, this can be avoided. Indeed, we will show that only examining the maximal distribution D_{max} is sufficient. For that, we will examine the semantics induced by two distributions D_1, D_2 in the distribution lattice $\langle \mathcal{D}_P, \preceq \rangle$ such that $D_1 \preceq D_2$ and show that the semantics induced by D_2 includes the one induced by D_1. Formally, we will prove that $K_{P,D_1} \lesssim K_{P,D_2}$. We will proceed in two steps. First, we will examine what happens if, starting from a distribution $D_1 \in \mathcal{D}_P$, one execution site of D_1 is split, leading to a distribution D_2. We will show that, in this case $K_{P,D_1} \lesssim K_{P,D_2}$ holds. Then, we will show that any $D_1, D_2 \in \mathcal{D}_P$ such that $D_1 \preceq D_2$ can be linked by a sequence of *splits* thus implying that $K_{P,D_1} \lesssim K_{P,D_2}$ holds in general. As a direct result, we will therefore get that model checking $K_{P,D_{\mathsf{max}}}$ is sufficient for any LTL-X property. In order to establish this result, we need three preliminary notions. The first one is that of *code distribution*.

Definition 3.9 (Code Distribution)

Let $\omega, \omega_1, \omega_2$ be blocks of code including internal instructions. We say that $\langle \omega_1, \omega_2 \rangle$ is a distribution of ω, noted $\omega \prec \langle \omega_1, \omega_2 \rangle$ if and only if one of the following holds:

(i) $(\omega = \varepsilon) \wedge (\omega_1 = \varepsilon) \wedge (\omega_2 = \varepsilon)$

(ii) $(\omega = x; \omega') \wedge (x \in \{\texttt{MSG}, \texttt{SEQ}\}) \wedge (\omega_1 = x; \omega'_1) \wedge (\omega_2 = x; \omega'_2) \wedge (\omega' \prec \langle \omega'_1, \omega'_2 \rangle)$

(iii) $(\omega = x; \omega') \wedge (x \notin \{\texttt{MSG}, \texttt{SEQ}\}) \wedge (\omega_1 = x; \omega'_1) \wedge (\omega' \prec \langle \omega'_1, \omega_2 \rangle)$

(iv) $(\omega = x; \omega') \wedge (x \notin \{\texttt{MSG}, \texttt{SEQ}\}) \wedge (\omega_2 = x; \omega'_2) \wedge (\omega' \prec \langle \omega_1, \omega'_2 \rangle)$

Intuitively, $\omega \prec \langle \omega_1, \omega_2 \rangle$ holds when all three blocks of code are empty, or when each instruction of ω is either in ω_1 or ω_2 unless it is a `MSG` or `SEQ` instruction, in which case, it must be present in both ω_1 and ω_2. The second notion is that of *valuation distribution*.

Definition 3.10 (Valuation Distribution)

Let X be a set of boolean variables, $X_1, X_2 \subseteq X$ be two subsets of X such that $X_1 \cup X_2 = X$, and three valuation $\nu \in X \mapsto \mathbb{B}$, $\nu_1 \in X_1 \mapsto \mathbb{B}$ and $\nu_2 \in X_2 \mapsto \mathbb{B}$. We say that $\langle \nu_1, \nu_2 \rangle$ is a distribution of ν, noted $\nu \prec \langle \nu_1, \nu_2 \rangle$, if and only if:

$$\forall x \in X : (x \in X_1 \Rightarrow \nu(x) = \nu_1(x)) \wedge (x \in X_2 \Rightarrow \nu(x) = \nu_2(x))$$

Intuitively, $\nu \prec \langle \nu_1, \nu_2 \rangle$ holds if the three valuation agree on the value of common variables. Finally, the last notion is that of *channel distribution*, which is very similar to code distribution.

Definition 3.11 (Channel Distribution)

Let $\sigma, \sigma_1, \sigma_2 \in ((V \times \mathbb{B}) \cup \{\diamond\})^*$ be the content of three communication channels. We say that $\langle \sigma_1, \sigma_2 \rangle$ is a distribution of σ if and only if one of the following holds:

(i) $(\sigma = \varepsilon) \wedge (\sigma_1 = \varepsilon) \wedge (\sigma_2 = \varepsilon)$

(ii) $(\sigma = \diamond \cdot \sigma') \wedge (\sigma_1 = \diamond \cdot \sigma'_1) \wedge (\sigma_2 = \diamond \cdot \sigma'_2) \wedge (\sigma' \prec \langle \sigma'_1, \sigma'_2 \rangle)$

(iii) $(\sigma = \langle x, v \rangle \cdot \sigma') \wedge (\sigma_1 = \langle x, v \rangle \cdot \sigma'_1) \wedge (\sigma' \prec \langle \sigma'_1, \sigma_2 \rangle)$

(iv) $(\sigma = \langle x, v \rangle \cdot \sigma') \wedge (\sigma_2 = \langle x, v \rangle \cdot \sigma'_2) \wedge (\sigma' \prec \langle \sigma_1, \sigma'_2 \rangle)$

Intuitively, $\sigma \prec \langle \sigma_1, \sigma_2 \rangle$ holds when all three channels are empty, or when each message of σ is either in σ_1 or σ_2 unless it is a $\diamond$ marker, in which case, it must be present in both σ_1 and σ_2. We can now prove the first step.

Lemma 3.2 (One-Split Simulation Lemma)

Given a well formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D_1, D_2 \in \mathcal{D}_P$ such that $D_1 \preceq D_2$ and $|D_2| = |D_1| + 1$, we have that:

$$K_{P,D_1} \lesssim K_{P,D_2}$$

Proof Sketch

First, we assume that $S = \{s_0, s_1, ..., s_n\}$, $K_{P,D_1} = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1 \rangle$, $K_{P,D_2} = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2 \rangle$ and, without loss of generality, that $D_1 = \{X_{1,1}, X_{1,2}, ..., X_{1,k}\}$ and $D_2 = \{X_{2,1}, X_{2,2}, ...X_{2,k-1}, X_{1,k}, X_{2,k+1}\}$ with $\forall i \in [1, k) : X_{1,i} = X_{2,i}$ and $X_{1,k} = X_{2,k} \cup X_{2,k+1}$. Then, we define a relation $S \subseteq Q_1 \times Q_2$ such that two states:

$$q_1 = \langle\, \langle \omega_{1,1}, \nu_{1,1}, \phi_{1,1} \rangle, ..., \langle \omega_{1,k}, \nu_{1,k}, \phi_{1,k} \rangle, \langle \sigma_{1,0}, \mu_{1,0} \rangle, ..., \langle \sigma_{1,n}, \mu_{1,n} \rangle \,\rangle$$
$$q_2 = \langle\, \langle \omega_{2,1}, \nu_{2,1}, \phi_{2,1} \rangle, ..., \langle \omega_{2,k+1}, \nu_{2,k+1}, \phi_{2,k+1} \rangle, \langle \sigma_{2,0}, \mu_{2,0} \rangle, ..., \langle \sigma_{2,n}, \mu_{2,n} \rangle \,\rangle$$

are related if and only if:

(i) $(\forall i \in [1, k) : \omega_{1,i} = \omega_{2,i}) \wedge (\omega_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle)$

(ii) $(\forall i \in [1, k) : \nu_{1,i} = \nu_{2,i}) \wedge (\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle)$

(iii) (a) $\forall j \in [1, k), \forall i \in [1, k) : \phi_{1,i \leftarrow j} = \phi_{2,i \leftarrow j}$
(b) $\forall j \in [1, k] : \phi_{1,k \leftarrow j} = \phi_{2,k \leftarrow j} = \phi_{2,k+1 \leftarrow j}$
(c) $\forall i \in [1, k] : \phi_{1,i \leftarrow k} \prec \langle \phi_{2,i \leftarrow k}, \phi_{2,i \leftarrow k+1} \rangle$

(iv) $\forall j \in [0, n] : (\sigma_{1,j} = \sigma_{2,j}) \wedge (\mu_{1,j} = \mu_{2,j})$

Then, it can be proven that S is a stuttering simulation relation between K_{P,D_1} and K_{P,D_2}. The complete proof is quite technical and requires some additional results. It is therefore presented separately in Appendix B.

The stuttering is required for instance to process the `MSG` marker in the workloads. Indeed, assume a process $P_{1,k}$ being split in two processes $P_{2,k}$ and $P_{2,k+1}$. Before this split, P_k can process a `MSG` marker in its workload in one transition, using Rule (3.5). After the split, however, there are two `MSG` markers to process, i.e. one in $P_{2,k}$ and one in $P_{2,k+1}$. In this case, two transitions are required. We now turn our attention to the second step, embodied in the following lemma.

Lemma 3.3

Given a well-formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D, D' \in \mathcal{D}_P$ such that $D \preceq D'$, we have that there exists a finite sequence $D_1, D_2, ..., D_n$ such that $D = D_1 \preceq D_2 \preceq ... \preceq D_n = D'$ and such that $\forall i \in [1, n) : |D_{i+1}| = |D_i| + 1$.

Proof

We proceed by induction on $|D'| - |D|$.

Initial Step If $|D'| - |D| = 0$ $(D = D')$ the sequence is empty.

Induction Step We know that $D \preceq D'$. Therefore, by Definition 3.4 of coarser/finer distribution, we have that $\forall X' \in D' : \exists X \in D : X' \subseteq X$, but since $|D'| > |D|$, by the pigeonhole principle $\exists X, Y \in D, \exists X' \in D' : (X \neq Y) \wedge (X \subseteq X') \wedge (Y \subseteq X')$. Let us construct $D'' = (D' \setminus \{X, Y\}) \cup \{X \cup Y\}$ by merging X and Y together. By construction, we have that $D'' \preceq D'$ with $|D'| = |D''| + 1$. Therefore, by induction, there exists a sequence $D = D_1, D_2, ..., D_n = D''$ from D to D''. The sequence from D to D' is then given by $D = D_1, D_2, ..., D_n, D_{n+1} = D'$.

Finally, we can conclude with the following theorem.

Theorem 3.3 (Stuttering Simulation of the Semantics)

Given a well formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D_1, D_2 \in \mathcal{D}_P$, we have that:

$$(D_1 \preceq D_2) \Rightarrow (K_{P,D_1} \lesssim K_{P,D_2})$$

Therefore, since stuttering simulation preserves LTL-X properties, this implies that if a LTL-X property holds on the semantics of a program induced by a given distribution, then it also holds on the semantics induced by any coarser distribution.

Corollary 3.1

Given a well-formed dSL program P, two distributions $D_1, D_2 \in \mathcal{D}_P$ such that $D_1 \preceq D_2$ and a LTL-X formula φ, we have that:

$$(K_{P,D_2} \models_{\text{L}} \varphi) \Rightarrow (K_{P,D_1} \models_{\text{L}} \varphi)$$

It follows directly that in order to prove that a dSL program P satisfies a LTL-X formula φ, one only has to examine the maximal distribution D_{max}.

Theorem 3.4 (dSL-LTL-MC: A Sufficient Condition for LTL-X Properties)

Given a well-formed dSL program P, and a LTL-X formula φ, we have that:

$$(K_{P,D_{\mathsf{max}}} \models_{\mathsf{L}} \varphi) \Rightarrow (P \models_{\mathsf{L}} \varphi)$$

Proof

By definition of D_{max}, we have that $\forall D \in \mathcal{D}_P : D \preceq \mathsf{D}_{\mathsf{max}}$. It follows by Corollary 3.1 that if $K_{P,D_{\mathsf{max}}} \models_{\mathsf{L}} \varphi$, then $\forall D \in \mathcal{D}_P : K_{P,D} \models_{\mathsf{L}} \varphi$. Hence, we conclude.

Note that Corollary 3.1 also implies that if a LTL-X property is violated for the minimal distribution D_{min}, then it is also violated for any distribution D of P.

3.3 Verification

As the reader may have noticed, even when restricted to a finite data domain, the semantics presented in Section 3.1.3 can yield *infinite-state* models. If we take a look at this semantics, we can quickly discover that two of the components in Definition 3.7 of global states are potentially unbounded in size, namely the *workloads* and the *communication channels*. Traditional model checking techniques however assume the underlying model to be *finite-state*. Nevertheless, a lot of research effort has been put in studying the model checking of *infinite-state* systems on various models. In particular, two models closely related to ours are of particular interest here. The first is that of *pushdown systems* [Bouajjani et al., 1997], i.e. system with potentially unbounded stacks. The other is that of *communicating finite state machines* (CFSM) [von Bochmann, 1978], i.e. systems communicating using potentially unbounded FIFO communication channels. For pushdown systems, even though they induce infinite-state models, the model checking problem is still decidable, both for LTL and CTL. More precisely, it is shown in [Bouajjani et al., 1997] that the model checking problem is DEXPTIME-complete for LTL and DEXPTIME-easy for CTL. This complexity was further refined in [Walukiewicz, 2000] where the model checking problem for CTL was shown PSPACE-complete. For the second class of systems, however, it was shown in [Brand and Zafiropulo, 1983], that CFSM are as expressive as *Turing Machines* [Turing, 1936], for which reachability, and therefore model checking, is known to be undecidable. Hence, a natural question then arises: *Is the model checking of* dSL *programs decidable?* Unfortunately, the answer to this question is *no*. Indeed, it turns out that CFSM can be simulated using dSL programs. However, as we will see, all hope is not lost.

The remainder of this section is structured as follows. First, in Section 3.3.1, we examine the model of CFSM in more details. Then, in Section 3.3.2, we show that CFSM can be simulated using dSL programs, therefore implying undecidability. We follow, in Section 3.3.3, by showing how the semantics can be constrained in order to avoid the undecidability. Finally, in Section 3.3.4, we show how this new constrained semantics can be used in practice to verify dSL programs.

3.3.1 Communicating Finite State Machine

Communicating finite state machines [von Bochmann, 1978], are systems with potentially unbounded FIFO communication channels. For our purpose here, we will restrict ourselves, without loss of generality, to CFSM's with only *one* communication channel, also called *message queue*. The formal definition follows.

Definition 3.12 (Communicating Finite State Machine)

A (single queue) communicating finite state machine (CFSM) is a tuple $M = \langle L, \ell_0, \Sigma, \Delta \rangle$ where:

- L is a finite set of control locations
- $\ell_0 \in S$ is the initial control location,
- Σ is a finite queue alphabet,
- $\Delta \subseteq L \times \{!, ?\} \times \Sigma \times L$ is the transition relation.

In this definition the message queue is implicit, since it is uniquely defined. Intuitively, a transition $\langle \ell, !, a, \ell' \rangle \in \Delta$ indicates that, when moving from location ℓ to location ℓ', a message $a \in \Sigma$ should be added at the end of the message queue. We call these transitions *write transitions*. On the other hand, a transition $\langle \ell, ?, a, \ell' \rangle$ indicates that, when moving from location ℓ to location ℓ', a message $a \in \Sigma$ should be present at the beginning of the message queue, and that it should be removed. We call these transitions *read* transitions. Formally, the semantics can be defined as follows.

Definition 3.13 (Semantics of Communicating Finite State Machine)

The semantics of a CFSM $M = \langle L, \ell_0, \Sigma, \Delta \rangle$ is a transition system $S_M = \langle Q, I, \rightarrow \rangle$ where:

- $Q \stackrel{\text{def}}{=} L \times \Sigma^*$ is the set of states,
- $I \stackrel{\text{def}}{=} \langle \ell_0, \varepsilon \rangle$ is the initial state,
- $\rightarrow$ is such that $\langle \langle \ell, w \rangle, \langle \ell', w' \rangle \rangle \in \rightarrow$ if and only if one of the following holds:
  - (i) $\exists \langle \ell, !, a, \ell' \rangle \in \Delta : w' = w \cdot a$
  - (ii) $\exists \langle \ell, ?, a, \ell' \rangle \in \Delta : a \cdot w' = w$

We say that a CFSM $M = \langle L, \ell_0, \Sigma, \Delta \rangle$ is *deterministic* if and only if $\forall \ell \in L, \forall c \in \{?, !\}, \forall a \in \Sigma : |\{\ell' \in L \mid \langle \ell, c, a, \ell' \rangle \in \Delta\}| \leq 1$. Note that this is only a syntactical definition. Indeed, it does not exclude the possibility of having two write transitions coming out of the same control location. Given a CFSM, $M = \langle L, \ell_0, \Sigma, \Delta \rangle$ and a control location $\ell \in L$, the *reachability problem* for CFSM (CFSM-REACH) consists in determining if $\exists w \in \Sigma^* : \langle \ell, w \rangle \in \mathsf{reach}(S_M)$. This problem is known to be undecidable.

Theorem 3.5 (Undecidability of CFSM-REACH [Brand and Zafiropulo, 1983])

The reachability problem for CFSM is undecidable.

This undecidability result also holds for deterministic CFSM's. Indeed, with the definition of determinism given above, one can determinize any CFSM using classical automata theory algorithms [Hopcroft et al., 2000].

3.3.2 Undecidability Result

In this section, we show that any deterministic CFSM can be simulated by a dSL program. More precisely, given a CFSM $M = \langle L, \ell_0, \Sigma, \Delta \rangle$, we build a well-formed dSL program P_M that simulates the behaviour of M. The construction works as follows. The current control location of M is encoded in P_M using $\lceil \log_2 |L| \rceil$ internal boolean variables `loc_b0`, `loc_b1`, ..., `loc_b`m, using a standard binary encoding. More precisely, assuming $L = \{\ell_0, \ell_1, ..., \ell_n\}$, a control location ℓ_i ($i \in [0, n]$) is encoded as a binary encoding of i. For the simplicity of the presentation, in the following, we will equivalently use an integer variable `loc` instead.

In a given control location ℓ, the decision of taking a particular transition, determined by an action (read or write) and a letter of Σ, is left to the environment of the program. The non-determinism of the environment will therefore ensure that every transition is explored. For this, $\lceil \log_2 |\Sigma| \rceil$ input variables `letter_b0`, `letter_b1`, ...,

`letter_b`r are needed to encode the letter chosen by the environment. More precisely, assuming $\Sigma = \{a_0, a_1, ..., a_s\}$, a letter a_i ($i \in [0, s]$) is encoded as a binary encoding of i. Again, for simplicity, in the following, we will equivalently use an integer variable `letter` instead. Another input boolean variable `readWrite` is then used to encode the action, i.e. whether that letter should be read or written. On top of that, one additional input boolean variable `tick` is also needed to trigger the transitions. Whenever the environment makes the variable `tick` switch from `FALSE` to `TRUE`, the program will consider the action proposed by the environment, i.e. the letter and the action that was chosen by the environment. If this action corresponds to a valid transition of M, the corresponding event is triggered and the variable `loc` is updated accordingly. Otherwise, the program simply ignores the choice proposed by the environment.

The only remaining thing for this construction to work, is to encode the message queue. This can be done using the FIFO communication channel defined by the semantics of P_M. Since M only uses *one* message queue, only one FIFO communication channel is needed, and therefore, only one execution site is required. The message queue is then modeled using the FIFO channel existing between this unique execution site and itself ($\phi_{0\leftarrow 0}$). Writing a letter a will be simulated using two consecutive assignments to a dedicated variable `a`. A first one to `FALSE` and a second one to `TRUE`. These assignments will cause their new value to be broadcast on the FIFO channel, thus effectively simulating the write. Reading a message can then simply be done using an event on $\sim$`a`[3]. When this event is triggered another dedicated internal boolean variable `aRead` is used to remember that fact.

Unfortunately, according to the semantics, any number of `dSL` messages can be treated during the processing phase, and this number, chosen non-deterministically, cannot be controlled in the program. It can therefore happen that the program deviates from the simulation. This can happen for several reasons:

(i) the environment wants the program to simulate a read transition on a letter a, whereas the non-determinism in the semantics is resolved in such a way that the two messages corresponding to the reception of that a have not been treated during the previous processing phase.

(ii) the environment wants the program to simulate a write transition, whereas the non-determinism in the semantics is resolved in such a way that the two messages corresponding to the reception of at least one letter have been treated during the previous processing phase.

[3]Remember indeed that, in `dSL`, a rising edge of the condition is needed to trigger an event.

(*iii*) the non-determinism in the semantics is resolved in such a way that messages corresponding to the reception of more than one letter have been treated since the last successful tick, i.e. the last time a transition was simulated.

However, these illegal behaviours correspond to particular assignments of the variables of the program and can therefore be detected. We therefore introduce an additional boolean variable `illegal`, that will be used to remember if and when such an illegal behaviour occurs. The event used to detect the reading of a letter a is presented in Figure 3.5(a). In this event, on top of the variable `aRead` being updated, another internal boolean variable `msgRead` is used to detect if more than one letter have been read since the last successful tick, i.e. the last time a transition was simulated by the program. In this case, as explained above, variable `illegal` is set to `TRUE` in order to remember that from now on, the dSL program deviates from the simulation.

The event used to simulate the transitions of M are presented in Figures 3.5(b) and (c). First, Figure 3.5(b) shows the event corresponding to a read transition $\langle \ell_1, ?, a, \ell_2 \rangle$. In this event, if an a was read since the last successful tick, the transition is taken, i.e. the current control location is updated (line 7) and the variables `aRead` and `msgRead` are reset for future use (lines 5–6). Otherwise, the variable `illegal` is set to `TRUE` (c.f. illegal behaviour (*i*) above). Figure 3.5(c) shows the event corresponding to a write transition $\langle \ell_1, !, a, \ell_2 \rangle$. In this event, if no message has been read since the last successful tick, i.e. the current control location is updated (line 7) and the send is simulated using two assignments to `a` (lines 5–6), as explained before. Otherwise, i.e. if at least one letter was read, the variable illegal is set to `TRUE` (c.f. illegal behaviour (*ii*) above). If at any point during the execution of the program, the variable `illegal` becomes `TRUE`, the simulation is blocked, since it is used in the test located at the beginning of each transition event. This effectively prevents any further transition from being taken. We can therefore conclude that only reachable locations of M will be reached in P_M. Formally, it can be easily established that a location $\ell \in L$ is reachable in M if and only if there exists a global state q, reachable in $K_{P_M, D_{\mathsf{min}}}$, where `loc`$= i$ holds. As a direct consequence, the LTL and CTL model checking problems are both undecidable for dSL programs.

Theorem 3.6 (Undecidability of Model Checking for dSL programs)
The LTL and CTL model checking problems for dSL programs are both undecidable.

```
1 WHEN ~a THEN
2   aRead := TRUE;
3   IF (NOT msgRead) THEN
4     msgRead := TRUE;
5   ELSE
6     illegal := TRUE;
7   END_IF;
8 END_WHEN
```

(a) Detecting when a letter a is read

```
1 WHEN tick THEN
2   IF (loc == l1) AND (readWrite == 0) AND
3      (letter == a) AND (NOT illegal) THEN
4     IF (aRead) THEN
5       aRead := FALSE;
6       msgRead := FALSE;
7       loc := l2;
8     ELSE
9       illegal := TRUE;
10    END_IF;
11  END_IF;
12 END_WHEN
```

(b) Simulating a read transition $\langle \ell_1, ?, a, \ell_2 \rangle$

```
1 WHEN tick THEN
2   IF (loc == l1) AND (readWrite == 1) AND
3      (letter == a) AND (NOT illegal) THEN
4     IF (NOT msgRead) THEN
5       a := 0;
6       a := 1;
7       loc := l2;
8     ELSE
9       illegal := TRUE;
10    END
11  END_IF;
12 END_WHEN
```

(c) Simulating a write transition $\langle \ell_1, !, a, \ell_2 \rangle$

Figure 3.5 - Events used to simulate a CFSM

```
1 VAR                          9 WHEN x THEN
2   x,y : BOOL;               10   x := FALSE;
3 END_VAR                     11   x := TRUE;
4                             12   y := TRUE;
5 SITE s                      13 END_WHEN
6   OUTPUT y : 1.0.0;         14
7 END_SITE                    15 SEQUENCE main()
                              16   x := TRUE;
                              17 END_SEQUENCE
```

Figure 3.6 - A dSL program with an infinite triggering loop

Proof

Given a deterministic CFSM $M = \langle L, \ell_0, \Sigma, \Delta \rangle$, and a control location $\ell \in L$, one can build a dSL program P_M using the construction explained above. Determining if $\exists w \in \Sigma^* : \ell \in \mathsf{reach}(T_M)$ can then be done by checking whether or not $K_{P_M, D_{\mathsf{min}}} \models_{\mathsf{L}} \mathsf{F}(\mathtt{loc} == \ell)$. However, according to Theorem 3.5, CFSM-REACH is undecidable. It follows directly that dSL-LTL-MC is also undecidable. A similar argument, using the formula $\mathsf{EF}(\mathtt{loc} == \ell)$ can be used for dSL-CTL-MC.

3.3.3 Constraining the Semantics

As we have seen previously, the model checking problem for dSL is undecidable both for LTL and CTL. Fortunately, in practice, all hope is not lost. Indeed, in practice, programs with unbounded *workloads* and/or *communication channel* should be avoided. Therefore, we can artificially bound the sizes of workloads and communication channels, thus leading back to a finite-state model. This will therefore allow us to apply standard finite-state model checking algorithms, and therefore use existing tools for verifying dSL programs.

For the workloads, only those included in the local states of processes are problematic. Those included in the local states of sequences are inherently bounded by the size of the corresponding sequence. For the workloads of the local states, the problem arises from the fact that the execution of the body of an event may trigger another event, and in particular, as illustrated by the program of Figure 3.6, it may trigger, directly or indirectly, itself. This causes an infinite control loop, where the workload of the process grows indefinitely. This kind of behaviour must of course be rejected. Indeed, in such a case, the process is stuck in its processing phase, and will therefore never reach its output phase again. For instance, in the program of Figure 3.6, the output variable `y` will never be assigned to `TRUE`, since the instruction `y := TRUE` will

never be reached in the workload. In practice, to prevent this from happening, the dSL virtual machine artificially bounds the number of imbricated triggered events.

Formally, the operational semantics presented in Section 3.1.3 can be modified to take this bound into account. For this, we need to determine the *triggering depth*, i.e. the number of events that have been triggered but for which the treatment is not finished yet. For that purpose, when handling an event e, we introduce a dummy assignment to the variable c_e ($c_e := c_e$ which has no side effects), inserted after the body of e. Using this trick, the triggering depth can be obtained at any point in a process, by counting the number of dummy assignments remaining in its workload. Formally, the treatment of events is modified as follows.

$$\mathsf{treat}(X) \stackrel{\text{def}}{=} \begin{cases} \left[\begin{array}{l} \texttt{IF } \mathsf{cond}(e) \texttt{ AND NOT } c_e \texttt{ THEN} \\ \quad c_e{:=}\texttt{TRUE}; \mathsf{body}(e); c_e{:=}c_e; \\ \texttt{ELSE} \\ \quad c_e{:=}\mathsf{cond}(e); \\ \texttt{END\_IF}; \mathsf{treat}(X \setminus \{e\})) \end{array}\right] & \text{if } X \neq \varnothing \wedge e = \mathsf{min}_{\lhd}(X) \\ \varepsilon & \text{if } X = \varnothing \end{cases}$$

The triggering depth can then be defined as follows.

Definition 3.14 (Triggering Depth)

Given a block of code ω the triggering depth in ω, noted $\mathsf{depth}(\omega)$, is defined inductively as follows.

$$\mathsf{depth}(\omega) \stackrel{\text{def}}{=} \begin{cases} \mathsf{depth}(\omega') + 1 & \text{if } \exists e \in E : \omega = c_e{:=}c_e; \omega' \\ \mathsf{depth}(\omega') & \text{if } \omega = i; \omega' \text{ such that } \forall e \in E : i \neq (c_e{:=}c_e) \\ 0 & \text{if } \omega = \varepsilon \end{cases}$$

Then, assuming the bound Θ on the triggering depth is at least 1, the only rules that need to be modified in the semantics are the two assignments rules corresponding to the handling of global variables. Indeed, these rules are the only ones that can lead to a triggering depth greater that 2. Those two rules need to be replaced by a bounded version. The first one is Rule (3.23) devised to replace Rule (3.8).

$$\frac{(\omega_i = x := v; \omega_i') \wedge (x \in X_i \cap V_{out}) \wedge (\mathsf{depth}(\omega_i') < \Theta)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_1, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \texttt{BCAST}(x, \mathsf{eval}[\nu_i](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega_i', \nu_i[\hat{x} \mapsto \mathsf{eval}[\nu_i](v)], \phi_i\rangle, ..., \ell_k, m_1, ..., m_n\rangle\end{array}} \quad (3.23)$$

```
1 VAR                          9 SITE s2
2   x, y : BOOL;              10   OUTPUT y : 1.0.0;
3 END_VAR                     11 END_SITE
4                             12
5 SITE s1                     13 WHEN ~x THEN
6   INPUT x : 1.0.0;          14   y := TRUE;
7 END_SITE                    15 END_WHEN
                              16
                              17 WHEN NOT ~x THEN
                              18   y := FALSE;
                              19 END_WHEN
```

Figure 3.7 - A dSL program with possibly unbounded channels channels

The other rule is Rule (3.24) devised to replace Rule (3.9).

$$\frac{(\omega_i = x := v; \omega_i') \wedge (x \in X_i \setminus V_{out}) \wedge (\mathsf{depth}(\omega_i') < \Theta)}{\begin{array}{c}\langle P, D\rangle \vdash \langle \ell_1, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \ell_k, m_1, ..., m_n\rangle \hookrightarrow \\ \langle \ell_1, ..., \langle \mathtt{BCAST}(x, \mathsf{eval}[\nu_i](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega_i', \nu_i[x \mapsto \mathsf{eval}[\nu_i](v)], \phi_i\rangle, ..., \ell_k, m_1, ..., m_n\rangle\end{array}} \quad (3.24)$$

Note that, with this modification, if the bound Θ is reached, process i will be blocked. Fortunately, static analysis techniques can be used to detect this does not happen, i.e. that the bound is never reached. Similarly to what is done in *bounded model checking* [Biere et al., 2003], the control flow graph [Aho et al., 1998] can be unfolded up to a depth of $\Theta + 1$, and then each path of this unfolding can be examined to check whether it is executable or not, i.e. check if the conditional statements can be traversed. For that purpose, a path can be encoded as a boolean formula in such a way that the formula is satisfiable if and only if the path is executable. This can be done using a classical *strongest post-condition* [Dijkstra and Scholten, 1990]. However, this method introduces a lot of quantifiers that makes the satisfiability checking more difficult. To avoid those unnecessary quantifications, *path simulation* [Ball and Rajamani, 2002] can be used instead. We implemented this procedure in the dSL compiler, using the theorem prover Simplify [Detlefs et al., 2003] for satisfiability checking. With this prototypical implementation, we were able to prove that the triggering depth was bounded in all examples we encountered. In fact, in most examples, the bound was quite small. Other techniques, based for instance on *interrupt calculus* [Chatterjee et al., 2003] could also be used.

For the communication channels, as illustrated in the program of Figure 3.7, the problem arises from the fact that one of the local processes can be significantly slower

that the other ones, in which case, its incoming communication channel can grow indefinitely. In the example, each time site `s1` is in its input phase, it sends a message to site `s2` with the value of the input variable `x`. The semantics, as defined in Section 3.1.3, allows a behaviour where `s2` is never scheduled, in which case, the messages in its incoming communication channels will never be treated. Similarly to what is done for the workloads, one can overcome this issue by artificially bounding the size of each communication channels. Again, the operational semantics can be modified in order to take care of this bound Γ. In this case, the only rule that needs to be modified is the broadcast rule, i.e. Rule (3.3). This rule is replaced by Rule (3.25) presented hereafter.

$$\frac{(\omega_i = \texttt{BCAST}(x, v); \omega_i') \wedge (\forall j \in [i, k] : |\phi_{i \leftarrow j}| < \Gamma)}{\begin{array}{c}\langle P, D\rangle \vdash \langle\langle \omega_1, \nu_1, \phi_1\rangle, ..., \langle \omega_i, \nu_i, \phi_i\rangle, ..., \langle \omega_k, \nu_k, \phi_k\rangle, m_1, ..., m_n\rangle \hookrightarrow \\ \langle\langle \omega_1, \nu_1, \phi_1'\rangle, ..., \langle \omega_i', \nu_i, \phi_i'\rangle, ..., \langle \omega_k, \nu_k, \phi_k'\rangle, m_1, ..., m_n\rangle\end{array}} \tag{3.25}$$

where $\forall j \in [1, k] : \phi_j' = \phi_j[i \mapsto \phi_{j \leftarrow i} \cdot \langle x, v\rangle]$.

In the following sections, we will therefore consider the semantics defined with those three modified rules. Given a dSL program P, a distribution D of P, and a pair of bounds Θ and Γ respectively on the size of workloads and communication channels, we note $K_{P,D}^{\Theta,\Gamma}$ the semantics of P w.r.t D bounded by Θ and Γ. Note that, even with this new bounded semantics, Theorem 3.3 still holds. Indeed, the stuttering simulation S used in Lemma 3.2 can also be used in this case. In fact, it is easy to see by definition of S that $\forall \langle q_1, q_2\rangle \in S$, if the sizes of the workloads, respectively communication channels, in q_1 are bounded by Θ, respectively Γ, then it is also the case in q_2. In fact, this can be extended to any $\Gamma' \geq \Gamma$ and $\Theta' \geq \Theta$. We can therefore extend Theorem 3.3 for the bounded semantics.

Theorem 3.7 (Stuttering Simulation of the Bounded Semantics)

Given a well formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda\rangle$, two distributions $D_1, D_2 \in \mathcal{D}_P$, and a pair Θ, Γ of bounds, we have that for any pair of bounds $\Gamma' \geq \Gamma$ and $\Theta' \geq \Theta$:

$$(D_1 \preceq D_2) \Rightarrow (K_{P,D_1}^{\Theta,\Gamma} \lesssim K_{P,D_2}^{\Theta',\Gamma'})$$

3.3.4 Verification in Practice

Using the bounded semantics defined in the previous section, we are now able to verify dSL programs. In this section we will concentrate on LTL properties, and show how

they can be verified using the model checker Spin [Holzmann, 2004].

Spin is one of the most successful and widely used model checkers for LTL. It is a model checker targeted towards *software* verification, as opposed to e.g. NuSMV [Cimatti et al., 1999; Cimatti et al., 2002] targeted towards *hardware* verification. It handles models expressed in a high-level description language called Promela (Process Meta Language). Promela is much like any common imperative language including local/global variables, conditional statements, loops, etc. It has also been enriched with non-determinism, inspired by Dijkstra's guarded command language [Dijkstra, 1975]. In Promela, models are described as a set of possibly dynamically instantiated processes (`proctype`) running in parallel, communicating with one another, either synchronously through *rendez-vous*, or asynchronously using *buffered message passing* or *shared variable* manipulation. Spin can handle efficiently very large models and has been successfully used to find logical errors in a large number of distributed systems, like e.g. operating systems, communication protocols, concurrent algorithms, etc. It enjoys a large user base, and exploits state-of-the-art technologies developed over the years. Spin's model checking algorithm is based on the *automata-based* approach proposed in [Vardi and Wolper, 1986], as explained in Chapter 1. The state space is built *on-the-fly*, thus allowing to only compute the portion of the state space that is needed to prove, or disprove, a property. In addition, Spin applies a technique called *partial order reduction* [Godefroid, 1996; Valmari, 1993]. This technique is based on the observation that, in concurrent systems, a given LTL formula is often insensitive to the order in which some concurrent actions occur. Therefore, only examining one of the possible interleavings of these actions is enough to decide whether or not this property holds on the model. This technique is very effective at reducing the size of the explored state space, and therefore the time and memory needed for model checking. Spin also allows the use of compact data structures, similar to BDDs [Bryant, 1992], to represent sets of states. It is worth mentioning however that this representation is used for *storage* purpose only. Indeed, the exploration is done through an *explicit-state* algorithm, meaning that each state in the set is explored individually, as opposed to a *symbolic-state* algorithm [McMillan, 1993], where the entire set is explored as a whole symbolically. For an in-depth description of Promela in particular, and Spin in general, we refer the reader to [Holzmann, 2004]. A short overview can also be found in [Holzmann, 1997].

The first thing to do, in order to verify LTL properties with Spin, is to translate the dSL program P into an equivalent Promela model. This can easily be done using the bounded semantics presented in the previous section. In fact, this semantics allows an

almost immediate translation. As expected, each global variable `x` of the dSL program is declared as a Promela variable. Moreover, an additional vector `tilde_x[N_SITES]` is declared for the tilded copies of `x`. Finally, for each event e, an internal `e_old_cond` (corresponding to c_e in the semantics) is declared to detect rising edges of the condition. The communication channels ϕ_i are directly translated in Promela as a vector of channels `phi_i[N_SITES]` of size Γ. Each local process in the semantics is translated into a Promela `proctype` executing its *input-process-output* behaviour. The input and output phases are left empty at first, and have to be written manually, depending on the environment. The processing phase starts with the treatment of all the events localized on site i. Then, a non deterministic number of messages from incoming communication channels are treated. Finally, sequences are treated. For this purpose, for each sequence s, two variables `s_site` and `s_pc` are declared to model respectively the site on which s is being currently executed and its program counter. The details of this translation are quite technical and will be therefore omitted here. However, for the interested reader, a simple example of dSL program and its corresponding Promela model is given in Appendix C.

Chapter

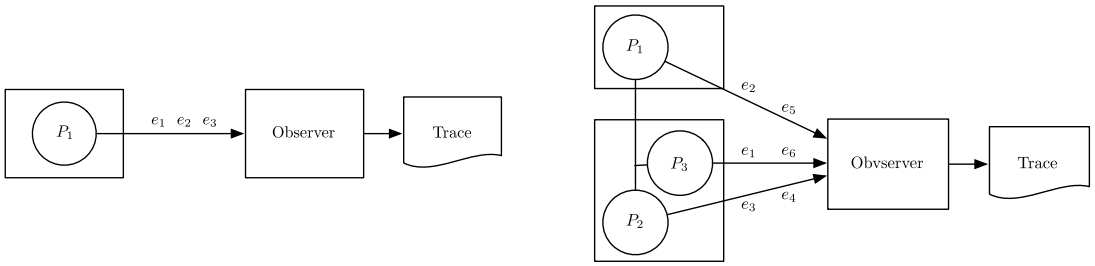
4

Testing dSL Programs

« *Beware of bugs in the above code;*
I have only proved it correct, not tried it. »

Donald Knuth

Model checking, as we have seen in Chapter 3, can be used to check temporal properties on dSL programs. However, due to the complexity of its semantics in general, and its distributed nature in particular, when dealing with large industrial applications, the designer is often faced with the unavoidable *state explosion problem*. This problem comes from the fact that the size of the model grows exponentially in the number of processes of the system. The size of the model therefore explodes and generally prevents the designer from exhaustively verifying the whole system, even with efficient exploration techniques such as *partial order reduction* [Godefroid, 1996; Valmari, 1993] or *symbolic* model checking [McMillan, 1993]. Furthermore, even if a complete model checking is possible, it is not a universal solution. Indeed, as its name indicates, model checking only works on a *model* of the system, which can differ from the *actual* system itself. This is particularly so when considering the environment, which is, more often than not, difficult to model accurately. The real environment often exhibits unpredicted behaviors that have not been taken into account in the model. Consequently, the absence of errors in the model does not necessarily imply the absence of errors in the actual system. If the model checking does not terminate, or if the designer simply wants additional validation on the actual system, he can turn to *testing*, which can unfortunately not guarantee that a system is completely bug-free, but if achieved on a large number of test-cases (e.g. covering most of the system's functionalities), can give a reasonable confidence in its correctness.



(a) Centralized system (b) Distributed system

Figure 4.1 - Centralised vs. Distributed system observation

The first step to test a property is *instrumentation*, i.e. the system under test is instrumented to emit events relevant for the given property[1]. This instrumented system is then executed, and the emitted events are observed by a separate process, called the *observer* or *monitor*. This observation is traditionally called a *trace*, which is analysed to determine whether or not the tested property is satisfied. This analysis can be done either offline, i.e. after the complete trace has been recorded, or online, i.e. while the system under test is running. When dealing with centralized systems, determining whether the property holds or not is generally quite simple. Indeed, in this case, all the events are emitted by a single process and are, as a consequence, totally ordered. In distributed systems, however, it is not that simple. Indeed, as illustrated in Figure 4.1, in the distributed case, the events are emitted by several processes. Two events emitted by concurrent processes are in general not always ordered. In a simple approach, one can assume that the events occurred in the order in which they are observed. However, this order does not necessarily coïncide with the actual order in which those events occurred. In the *predictive* approach [Sen et al., 2004a], communications between concurrent processes are exploited in the instrumentation step, in order to obtain a partial order, instead of a total order. In this approach, the analysis consists in determining if the property holds for every total order compatible with the observation.

In this chapter, we examine how to apply this approach to test distributed concurrent systems in general, and dSL programs in particular. We start, in Section 4.1, by explaining how a partial order on the events of the system can be obtained using *vector clocks* [Mattern, 1989]. Then, in Section 4.2, we introduce a formal model to capture this partial order, namely the model of *partial order traces*. We then explain in Section 4.3, how to specify properties on this model using classical formal logics.

[1]The notion of event here is to be understood in a larger sense than in Chapters 2 and 3

We examine both non-temporal and temporal properties and, in each case, formalize the corresponding analysis problem, i.e. determining if a given partial order trace satisfy a given property. Finally, we conclude this chapter, in Section 4.4, with a short discussion. The content of this chapter is based on a joint work with Thierry Massart and Laurent Van Begin, accepted for publication [Massart et al., 2007].

4.1 Instrumentation

In a centralized system, it is straightforward to tell if an event precedes another one, since all events are emitted by the same process. A simple logical clock, counting the number of events, can therefore be used to time-stamp each event. Determining the order of two events can then simply be done by comparing the values of their respective clocks. In a distributed system, however, only events emitted by the same process, are ordered, while concurrent events are, in general, not. There is one exception where additional ordering information can be obtained, that is when the concurrent processes communicate. In this case, as mentioned in the introduction, the communication scheme can be used to obtain a partial order on the events of the system. In practice, vectors of logical clocks, called *vector clocks*, can be used to time-stamp the events of the system. Determining the order of two events then amounts to a component-wise comparison of their respective vector clocks. If the clocks are incomparable, like e.g. $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$, then this means that the events are concurrent, i.e. the exact order in which they occurred cannot be determined. Formally, vector clocks are defined as follows.

Definition 4.1 (Vector Clocks)

Given a poset $\langle X, \sqsubseteq\rangle$, a vector clock mapping of width n is a function $vc \in X \mapsto \mathbb{N}^n$, such that:

$$\forall x, x' \in X : (x \sqsubseteq x') \Leftrightarrow (vc(x) \leq vc(x'))$$

In practice, for a distributed system composed of k processes, the partial order on events is represented using a vector clock mapping of width k. The method for computing this vector clock mapping depends on the communication scheme used in the system. As we have seen in Chapter 3, systems programmed in dSL communicate via message passing. Therefore, we review, in Section 4.1.1 an algorithm for message passing programs. However, we want the techniques and models used in this work to be as general as possible. That is why, we also review, in Section 4.1.2, an algorithm for programs manipulating shared variables.

```
input  : an event e of process P_i, the current vector clock v_i ∈ ℕ^k of P_i
output : the vector clock v_i after the occurrence of e
1  begin
2      if e is a relevant event then
3          v_i[i] := v_i[i] + 1
4      if e is the sending of a message m to process P_j then
5          send ⟨m, v_i⟩ to P_j instead of m
6      else if e is the reception of a message ⟨m, v⟩ then
7          forall j ∈ [1, k] do
8              v_i[j] := max_≤(v_i[j], v_m[j])
9      if e is a relevant event then
10         send ⟨e, v_i⟩ to the observer
11 end
```

Algorithm 4.1 - Mattern's vector clock algorithm for message passing program

4.1.1 Message Passing Programs

In a system where the processes communicate asynchronously via message passing, we can ascertain that the sending of a message always precedes its reception. Using this additional information, a partial order, known as the *happened-before* relation [Lamport, 1978], can be obtained. This relation is defined as follows.

Definition 4.2 (Happened-Before Relation [Lamport, 1978])

The *happened-before* relation is the smallest transitive binary relation on the events of the systems satisfying the following two conditions:

(i) if e and e' are events emitted from the same process, and if e comes before e', then and e *happened-before* e'

(ii) if e is the sending of one message by one process, and e' is the reception of that same message by another process, then e *happened-before* e'.

The vector clock mapping for this *happened-before* relation can be obtained using the well-known algorithm by Mattern [Mattern, 1989]. In this algorithm, each process P_i maintains its own vector clock $v_i \in \mathbb{N}^k$, where k is the number of processes in the system. At any given time during the execution of the program, $v_i[j]$ is used to store the number of events from process P_j that process P_i is aware of, i.e. the number of events

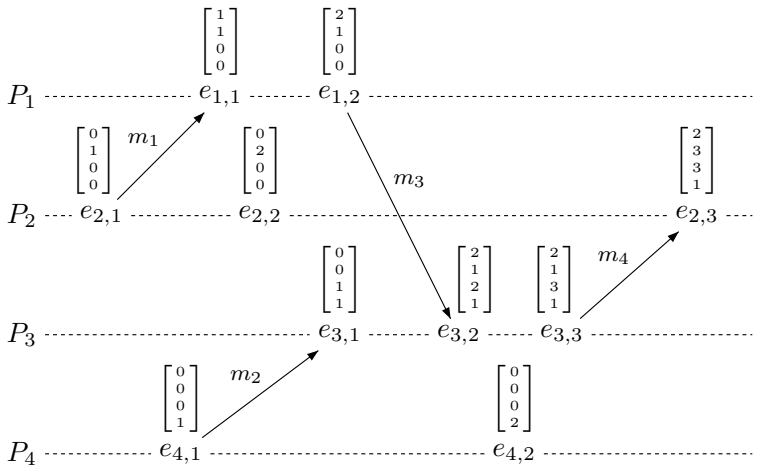


Figure 4.2 - Execution of a message passing program with vector clock mapping

that *happened-before* the most recent event of process P_i. In particular, $v_i[i]$ gives the number of events of process P_i that already occurred. Initially, each component of v_i is set to 0. Then, when an event e occurs on process P_i, its vector clock v_i is updated as described in Algorithm 4.1. First, if e is a relevant event, then $v_i[i]$ is incremented, indicating that a new event as occurred in process P_i (lines 2–3). Then, if e corresponds to the sending of a message m, the vector clock v_i is piggybacked along with it (lines 4–5). On the other hand, if e corresponds to the reception of a message m, tagged with a vector clock v_m, then v_i is set to the component-wise maximum of v_i and v_m, to take into account the fact that any event preceding the sending of m should also precede e (lines 6–8). Finally, after this update, again if e is relevant, a message $\langle e, v_i \rangle$ is sent to the observer (lines 9–10). The vector clock mapping vc is then built by the observer using these messages. More precisely, we have that for any message $\langle e, v \rangle$ received by the observer, $vc(e) \stackrel{\text{def}}{=} v$.

Example 4.1

Figure 4.2 shows a sample execution, taken from [Mattern, 1989], of a message passing program with 4 processes, and the vector clock mapping obtained using Mattern's algorithm. In this diagram, $e \xrightarrow{m} e'$ indicates that e is the sending of a message m, and e' is its reception. Consider, for instance, events $e_{4,1}$ and $e_{2,3}$. Those events are ordered since $e_{4,1} \xrightarrow{m_2} e_{3,1}$, $e_{3,1}$ comes before $e_{3,3}$ on process P_3, and $e_{3,3} \xrightarrow{m_4} e_{2,3}$. Note that their respective vector clocks are also ordered.

This algorithm was later extended by Fidge to account for dynamic process creation. We refer the reader to [Fidge, 1991] for more details on that.

input : an event e of process P_i, the current vector clock $v_i \in \mathbb{N}^k$ of P_i, the access/write vector clocks $v_{a(x)}/v_{w(x)}$ for each shared variable x
output : the vector clock v_i after the occurrence of e

```
1  begin
2      if e is a relevant event then
3          v_i[i] := v_i[i] + 1
4      forall shared variable x that is read by e do
5          forall j ∈ [1, k] do
6              v_i[j] := max_≤(v_i[j], v_w(x)[j])
7              v_a(x)[j] := max_≤(v_i[j], v_a(x)[j])
8      if e is a write to a shared variable x then
9          forall j ∈ [1, k] do
10             v_i[j] := max_≤(v_i[j], v_a(x)[j])
11         v_w(x) := v_a(x) := v_i
12     if e is a relevant event then
13         send ⟨e, v_i⟩ to the observer
14 end
```

Algorithm 4.2 - Sen's vector clock algorithm for multithreaded program

4.1.2 Shared Variable Programs

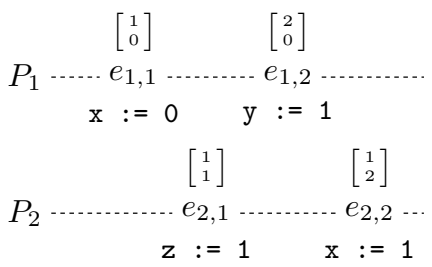
In a system where processes communicate by manipulating shared variables, a similar idea can be applied. In this case, we can ascertain that each *write* to a shared variable x should precede every subsequent *access* to x (read or write). Using this idea, a vector clock algorithm was proposed by Sen et al. [Sen et al., 2003]. For that purpose, two additional vector clocks $v_{a(x)}, v_{w(x)} \in \mathbb{N}^k$ are introduced for each shared variable x of the program. Intuitively $v_{a(x)}$, respectively $v_{w(x)}$, is the vector clock corresponding to the most recent *access*, respectively *write*, to a shared variables x. At any given time during the algorithm, $v_{a(x)}[j]$, respectively $v_{w(x)}[j]$, is used to store the number of events of process P_j that precedes the most recent event of process P_i that *accessed*, respectively *wrote* to, variable x. On top of that, similarly to Mattern's algorithm, each process P_i maintains its own vector clock $v_i \in \mathbb{N}^k$. Initially, each component of v_i, as well as $v_{a(x)}$ and $v_{w(x)}$ for each shared variable x, is set to 0. When an event e occurs on process P_i, the vector clock v_i and the vector clocks of the shared variables

```
var x = -1;
var y = 0;
var z = 0;

process P1() {
   x := x + 1;
   y := x + 1;
}

process P2() {
   z := x + 1;
   x := x + 1;
}
```

(a) The program



(b) The execution

Figure 4.3 - Execution of a shared variable program with vector clock mapping

are updated as described in Algorithm 4.2. First, if e is a relevant event, then $v_i[i]$ is incremented, as in the message passing case (lines 2–3). Then for each shared variable x that is read by e, the vector clock v_i is set to the component-wise maximum of v_i and $v_{w(x)}$ (line 6), to take into account the fact that last event that wrote to x should precede e. Moreover, since e is an access to x, the vector clock $v_{a(x)}$ is updated with the component-wise maximum of v_i and $v_{a(x)}$ (line 7). Next, if e is a write to a shared variable x, then the vector clock v_i is set to the component-wise maximum of v_i and $v_{a(x)}$ (lines 9–10), so as to force e to precede any future access to x. Moreover, since e is both a write and an access to x, the two corresponding vector clocks are updated with v_i (line 11). Finally, after this update, similarly to the message passing case, the message $\langle e, v_i \rangle$ is sent to the observer, which can then build the vector clock mapping for the execution.

Example 4.2

Figure 4.3(b) shows a sample execution, taken from [Sen et al., 2003], of the shared variables program of Figure 4.3(a) with 2 processes, and the vector clock mapping obtained using the algorithm presented above. In this diagram, each event e is decorated with the observed assignment. Consider, for instance, events $e_{1,1}$ and $e_{2,2}$. Those events are ordered since $e_{1,1}$ assigns the shared variable x, which is used in the assignment of $e_{2,1}$ (`z := 1`), and since $e_{2,1}$ comes before $e_{2,2}$ on process P_2. Notice that their respective vector clocks are also ordered.

4.2 Partially Ordered Trace

Using the instrumentation technique presented in the previous section, the observer can build the partial order corresponding to a distributed execution. The next step consists in analyzing this partial order execution in order to determine whether it satisfies a given property. For that purpose, we introduce the formal model of *partial order trace*, po-trace for short, where events are abstracted as predicate transformers, i.e. atomic actions modifying the truth value of one or more boolean propositions.

4.2.1 Definition

Essentially, a po-trace is a poset of events partitioned into several subsets called processes, along with an update function indicating for each event e, which propositions are modified by e, as well as how they are modified. Furthermore a po-trace also includes the set of propositions that are initially true. The formal definition follows.

Definition 4.3 (Partial Order Trace)

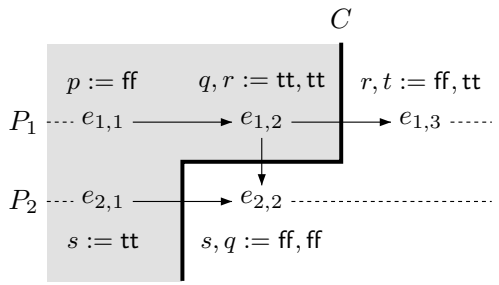
A partial order trace T over a set of propositions $\mathbb{P}$ is a tuple $\langle E, V_0, \delta, \preceq \rangle$ where:

- E is a finite set of events partitioned into k disjoint subsets P_i;
- $V_0 \subseteq \mathbb{P}$ is the set of propositions initially true;
- $\delta \in E \times \mathbb{B} \mapsto 2^{\mathbb{P}}$ is the update function s.t. $\forall e \in E : \delta(e, \mathsf{tt}) \cap \delta(e, \mathsf{ff}) = \varnothing$,
- $\preceq \subseteq E \times E$ is a partial order on E such that $\forall e, e' \in E$ the following holds:

  (i) $(\exists i \in [1,k] : \{e, e'\} \subseteq P_i) \Rightarrow (e \preceq e' \vee e' \preceq e)$

  (ii) $((\delta(e, \mathsf{tt}) \cup \delta(e, \mathsf{ff})) \cap (\delta(e', \mathsf{tt}) \cup \delta(e', \mathsf{ff})) \neq \varnothing) \Rightarrow (e \preceq e' \vee e' \preceq e)$

In the previous definition, a proposition $p \in \delta(e, b)$ indicates that e sets the truth value of p to b. Condition (i) on $\preceq$ ensures that two events belonging to the same process should be ordered and condition (ii) enforces that if the truth value of at least one proposition is modified by two events, then those events should be ordered as well. Note that, because of condition (i) in Definition 4.3, for any process $P_i \subseteq E$, $\langle P_i, \preceq \rangle$ is also a totally ordered set. Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over a set of propositions $\mathbb{P}$, a set of events $F \subseteq E$, and a proposition $p \in \mathbb{P}$, we define the set of events of F that affect the truth value of p as $F_{/p} \stackrel{\text{def}}{=} \{e \in F \mid p \in \delta(e, \mathsf{tt}) \cup \delta(e, \mathsf{ff})\}$. Note that $F_{/p} = F \cap E_{/p}$. Because of condition (ii) in Definition 4.3, for any subset $F \subseteq E$ and $p \in \mathbb{P}$, $\langle F_{/p}, \preceq \rangle$ is a totally ordered set. If there is only one process, we say that T is a total order trace since $\langle E, \preceq \rangle$ is a totally ordered set by condition (i). The size of a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ is defined as $|T| \stackrel{\text{def}}{=} \mathsf{max}_{\leq}(\{|E|, |\preceq|\})$.



with $V_0 = \{p\}$

Figure 4.4 - Example of po-trace

Example 4.3

Figure 4.4 shows a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over the set of propositions $\{p, q, r, s\}$. Events are depicted in the Hasse diagram of the poset $\langle E, \preceq \rangle$, and are decorated with multiple variable assignments to represent δ. For instance, if we consider the event $e_{1,2}$, "$q, r{:=}\ \mathsf{tt}, \mathsf{tt}$" indicates that $\delta(e_{1,2}, \mathsf{tt}) = \{q, r\}$ and that $\delta(e_{1,2}, \mathsf{ff}) = \varnothing$. In this picture, a cut $C = \{e_{1,1}, e_{1,2}, e_{2,1}\}$ has been highlighted in gray.

4.2.2 Semantics

The semantics of a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ is given as a Kripke structure K_T. Each state of K_T is a downward closed subset of events called a *cut*, or *global state*. In this Kripke structure, each cut is labelled with the set of propositions that are true after executing the events it contains. The formal definition follows.

Definition 4.4 (Semantics of Partial Order Traces)

The semantics of a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$, is a Kripke structure $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ where:

- $Q \stackrel{\text{def}}{=} \mathsf{DC}_{\preceq}(E)$
- $I \stackrel{\text{def}}{=} \{\varnothing\}$
- $\mathcal{L} \stackrel{\text{def}}{=} \lambda C \cdot \{p \in \mathbb{P} \mid (C_{/p} = \varnothing \wedge p \in V_0) \vee (C_{/p} \neq \varnothing \wedge p \in \delta(\mathsf{max}_{\preceq}(C_{/p}), \mathsf{tt})\}$
- $\rightarrow \stackrel{\text{def}}{=} \{\langle C, C \cup \{e\}\rangle \in Q \times Q \mid (C \in Q) \wedge (e \in E \setminus C)\}$

In the previous definition, a propositions $p \in \mathbb{P}$ is contained in $\mathcal{L}(C)$ either (i) if C does not contain any event updating the truth value of p ($C_{/p} = \varnothing$) but p is initially true ($p \in V_0$), or (ii) if C contains some event(s) updating the truth value of p ($C_{/p} \neq \varnothing$) and the most recent one ($\mathsf{max}_{\preceq}(C_{/p})$) sets p to true. A *local state* of a

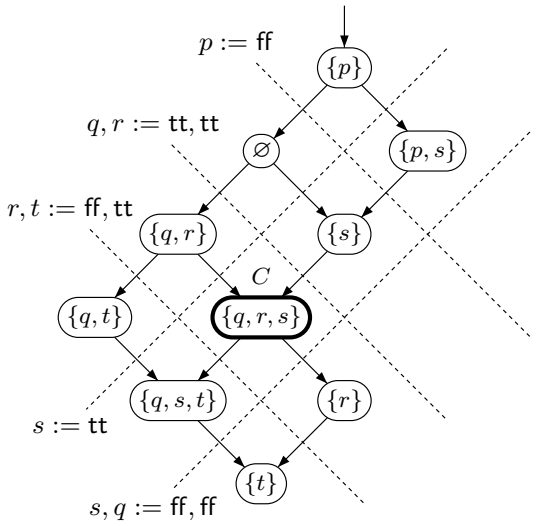


Figure 4.5 - Semantics of the po-trace of Figure 4.4

process P_i is a subset $S_i \subseteq P_i$ such that $\downarrow S_i \cap P_i = S_i$. Given a cut $C \in Q$, the *local state* of process P_i in C is defined as $P_i \cap C$. Moreover, the set of events *enabled* in C is defined as $\mathsf{enabled}(C) \stackrel{\text{def}}{=} \{e \in E \setminus C \mid \downarrow e \setminus \{e\} \subseteq C\}$. Intuitively, $\mathsf{enabled}(C)$ gives the set of events that can be triggered from the cut C. This is formalized hereafter.

Proposition 4.1

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we have that:

$$\forall C, D \in Q : (C \rightarrow D) \Leftrightarrow (\exists e \in \mathsf{enabled}(C) : D = C \cup \{e\})$$

Proof

We prove both direction:

- $\boxed{\Rightarrow}$ Assume that $C \rightarrow D$. By Definition 4.4, we know that $\exists e \in E \setminus C$ such that $D = C \cup \{e\}$. Moreover, since $D \in Q = \mathsf{DC}_{\preceq}(E)$ and since $e \in D$, we have that $\downarrow e \subseteq D$. It follows directly that $\downarrow e \setminus \{e\} \subseteq D \setminus \{e\} = C$. Therefore, by definition of $\mathsf{enabled}(C)$, we have that $e \in \mathsf{enabled}(C)$.
- $\boxed{\Leftarrow}$ Assume the existence $e \in \mathsf{enabled}(C)$ such that $D = C \cup \{e\}$. By hypothesis, we know that $C \in Q = \mathsf{DC}_{\preceq}(E)$. We also know, by definition of $\mathsf{enabled}(C)$ that $\downarrow e \setminus \{e\} \subseteq C$. It follows directly that $\downarrow e \subseteq C \cup \{e\}$. Hence, we have that $C \cup \{e\} = D \in \mathsf{DC}_{\preceq}(E) = Q$, and by Definition 4.4, that $C \rightarrow D$.

Example 4.4

Figure 4.5 shows the semantics of the po-trace of Figure 4.4, where, the triggerings of events are indicated with dashed lines. Furthermore, the cut C, highlighed in gray in Figure 4.4 is highlighted in bold. In this cut, we have that $\mathsf{enabled}(C) = \{e_{1,3}, e_{2,2}\}$. Finally, the local state of process P_1, respectively P_2, in C is $\{e_{1,1}, e_{1,2}\}$, respectively $\{e_{2,2}\}$.

In the semantics of a po-trace, the reflexive and transitive closure of the transition relation coincides with set inclusion.

Proposition 4.2

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we have that:

$$\forall C, D \in Q : (C \subseteq D) \Leftrightarrow (C \rightsquigarrow D)$$

Proof

We prove both directions.

$\boxed{\Rightarrow}$ Assume that $C \subseteq D$. We prove by induction on $|D \setminus C|$ that $C \rightsquigarrow D$.

Initial Step If $|D \setminus C| = 0$, we have that $C = D$ and the result is immediate.

Induction Step If $|D \setminus C| = n > 0$, we have that $D \setminus C \neq \varnothing$. Let $e \in \mathsf{Min}_{\preceq}(D \setminus C)$ be a minimal elements of $D \setminus C$. We know that $e \in D \in Q = \mathsf{DC}_{\preceq}(E)$. Therefore, we have that $\downarrow e \subseteq D$. We also know that $e \in \mathsf{Min}_{\preceq}(D \setminus C)$, which implies that $\forall e' \in D \setminus C : (e' \preceq e) \Rightarrow (e = e')$. Therefore, it must be that $\forall e' \in E : \big((e' \preceq e) \wedge (e' \neq e)\big) \Rightarrow (e' \in C)$. It follows directly that $e \in \mathsf{enabled}(C)$, and by Propositions 4.1, that $C \rightarrow C \cup \{e\}$. Finally, since $C \cup \{e\} \subseteq D$, by induction, we have that $C \cup \{e\} \rightsquigarrow D$, which in turn, implies that $C \rightsquigarrow D$.

$\boxed{\Leftarrow}$ Assume on the other hand, that $C \rightsquigarrow D$. By definition of $\rightsquigarrow$, there exists a sequence of cut $C_0 C_1 \ldots C_n$ such that $C = C_0$, $C_n = D$ and that $C_i \rightarrow C_{i+1}$ for any $i \in [0, n)$. By Definition 4.4, it follows that for any $i \in [0, n)$, we have $C_i \subseteq C_{i+1}$. Finally, by transitivity of $\subseteq$, we can conclude that $C = C_0 \subseteq C_n = D$.

Note, in particular, that given a cut $C \in Q$, since $\varnothing \subseteq C \subseteq E$, we have that $\varnothing \rightsquigarrow C \rightsquigarrow E$, or in other words that C belongs to at least one run of K_T. In fact, $\langle Q, \rightsquigarrow \rangle$ is a complete distributive lattice.

Theorem 4.1 (Lattice of Cuts)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we have that $\langle Q, \rightsquigarrow \rangle$ is a complete distributive lattice.

Proof

We know, by Definition 4.3, that E is finite, hence by Theorem 1.1, we have that $\langle \mathsf{DC}_{\subseteq}(E), \subseteq \rangle = \langle Q, \subseteq \rangle$ is a complete distributive lattice. Finally, using Propositions 4.2, we know that $\rightsquigarrow$ and $\subseteq$ are equivalent on Q. Hence, we conclude.

4.3 Specification

In Section 4.1, we have seen how to instrument a distributed system in order to obtain a partial order on the events of this system. We also introduced, in Section 4.2, a model to capture this partial order, namely partial order traces. We now explain how to exploit this model in order to specify properties using classical formal logics and define the associated trace analysis problem. First, in Section 4.3.1, we focus our attention on non-temporal properties and explain how they can be specified using PBL formulae. Next, in Sections 4.3.2 and 4.3.3, we turn our attention to temporal properties expressed respectively in LTL and CTL.

4.3.1 Non Temporal Properties

Non temporal properties are essentially reachability questions, i.e. during a distributed execution, does the system go through a global state satisfying some given constraint called a *predicate*. This problem is known as the *predicate detection problem*. In our framework, since partial order traces are defined over boolean propositions, those predicates can be expressed using PBL formulae. Formally, given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a PBL formula φ, we say that T satisfies φ, noted $T \models_{\mathsf{P}} \varphi$, if and only if $[\![\varphi]\!]^T_{\mathsf{P}} \neq \varnothing$, where $[\![\varphi]\!]^T_{\mathsf{P}} = \{C \in Q \mid \mathcal{L}(C) \models_{\mathsf{P}} \varphi\}$. We can then formalize the predicate detection problem as follows.

Definition 4.5 (Predicate Detection Problem)

Given a po-trace T over a set of propositions $\mathbb{P}$ and a PBL formula φ over $\mathbb{P}$, the predicate detection problem (PRED) consists in determining if $T \models_{\mathsf{P}} \varphi$.

This predicate detection problem is not an easy problem. Indeed, the number of cuts in a given po-trace can be exponentially bigger than the number of events in this trace. In fact, it was proven in [Chase and Garg, 1995] that this exponential blowup cannot be avoided. More precisely, the authors proved that the predicate

detection problem is NP-complete. We adapt the proof of [Chase and Garg, 1995] to our framework. First we show that PRED is in NP.

Lemma 4.1 ([Chase and Garg, 1995])

The predicate detection problem is in NP.

Proof

We provide a non-deterministic polynomial algorithm. This algorithm works as follows. First it guesses a subset $C \subseteq E$. Then, it checks that $C \in Q$. This can be done in a time linear in $|\preceq| \leq |T|$, by checking for every pair $\langle e, e' \rangle \in \preceq$ that if $e' \in C$, then $e \in C$. Finally, the algorithm checks that $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$. This can be done in a time linear in $|\varphi|$ and in $|C| \leq |T|$ (for building $\mathcal{L}(C)$).

Then, we prove NP-hardness.

Lemma 4.2 ([Chase and Garg, 1995])

The predicate detection problem is NP-hard.

Proof

We give a reduction from PBL-SAT. Given a PBL formula φ, we build a po-trace $T_\varphi = \langle E, V_0, \delta, \preceq \rangle$ over $\mathsf{prop}(\varphi)$ as follows: (i) T_φ has one process $P_p \stackrel{\text{def}}{=} \{e_p\}$ for each proposition p appearing in φ, i.e. $E \stackrel{\text{def}}{=} \bigcup_{p \in \mathsf{prop}(\varphi)} P_p$, (ii) the initial valuation is empty, i.e. $V_0 = \varnothing$, (iii) each event e_p sets the corresponding proposition p to tt, i.e. $\delta(e_p, \mathsf{tt}) \stackrel{\text{def}}{=} \{p\}$ and $\delta(e_p, \mathsf{ff}) = \varnothing$, and finally (iv) all events are concurrent, i.e. $\preceq \stackrel{\text{def}}{=} \varnothing$. Then, assuming that $K_{T_\varphi} = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we prove that φ is satisfiable if and only if $T \models_{\mathsf{P}} \varphi$. We prove both directions.

- $\boxed{\Rightarrow}$ Assume the existence of $V \subseteq \mathbb{P}$ such that $V \models_{\mathsf{P}} \varphi$. Let us build a cut $C \stackrel{\text{def}}{=} \{e_p \in E_\varphi \mid p \in V\}$. By Definition 4.4 of the semantics of po-traces, we have that $\mathcal{L}(C) = V$. It follows directly that $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$. We can therefore conclude that $[\![\varphi]\!]^T_{\mathsf{P}} \neq \varnothing$ and therefore that $T \models_{\mathsf{P}} \varphi$.
- $\boxed{\Leftarrow}$ Assume the existence of $C \in Q$ such that $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$. We known that $\mathcal{L}(C) \subseteq \mathbb{P}$, we can therefore directly conclude that φ is satisfiable.

It follows directly that the predicate detection problem is NP-complete.

Theorem 4.2 (Complexity of PRED [Chase and Garg, 1995])

The predicate detection problem is NP-complete.

Note that this NP-completeness arises from the distributed nature of our traces. Indeed, on total order traces, the predicate detection problem can be solved in polynomial time.

Theorem 4.3 (Complexity of PRED for Total Order Traces)

The predicate detection problem is in PTIME for total order traces.

Proof

A total order trace T admits exactly one run ρ, since its events are totally ordered. Moreover, by Definition 4.4, we have that $|\rho| = |E|+1$. This run therefore contains only a polynomial number of cuts. Therefore, the predicate detection problem can be solved by examining the cuts of the run and for each such cut C, determining if $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$. For that, one has to compute the valuation $\mathcal{L}(C)$, which can be done in polynomial time in $|C| \leq |T|$ by examining the events of C, and determine if this valuation satisfies φ, which can be done in polynomial time in $|\varphi|$.

4.3.2 LTL Properties

We have seen in the previous section that non temporal properties are reachability questions, i.e. does there exists a cut of the trace in which a given PBL formula holds. Such reachability questions can easily be expressed in LTL using the G modality. More precisely, given a po-trace T and a PBL formula φ, we could equivalently ask if $\mathsf{G}\,\neg\varphi$ is satisfied by T, i.e. if for all runs of T, φ is never satisfied. Indeed, if the answer to this question is positive, that means that T does not contain any cut satisfying φ, i.e. $T \not\models_{\mathsf{P}} \varphi$. However, if we want to do that, we need to formalize what it means for a partial order trace to satisfy a LTL formula. This is not as trivial as it may seem at a first glance. Indeed, even though the semantics of partial order traces is defined in terms of Kripke structures, the traditional semantics of LTL, that was presented in Chapter 1, cannot be used. This semantics was defined in the context of model checking, in which execution traces are *infinite* propositional sequences that are obtained from a Kripke structure modeling the *entire* system. In our case, however, the Kripke structure comes from the semantics of a po-trace which represents only a set of *finite* propositional sequences. We therefore need to adapt the classical semantics to account for the finite nature of partial order traces.

Intuitively, in the infinite trace semantics, a LTL formula contains a safety part and/or a liveness part. The safety part of the formula states that something bad never happens. On the other hand, the liveness part states that something good will eventually happen. In our context of finite propositional sequences, we need to modify this interpretation so that it states that something good/bad eventually/never happens *within the finite portion of the execution that has been observed*. This means, for instance, that in every formula of the form $\varphi\,\mathsf{U}\,\psi$, the right branch must be satisfied before the end of the propositional sequence. The formal definition follows.

Definition 4.6 (Semantics of LTL over Finite Sequences)

Given a set of propositions $\mathbb{P}$, let φ, ψ be two LTL formulae over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition, $\sigma \in (2^{\mathbb{P}})^+$ be a finite non-empty propositional sequence over $\mathbb{P}$ and $i \in [0, |\sigma|)$ be a natural number. The satisfaction relation for LTL over finite propositional sequences, noted $\models^{\mathsf{F}}_{\mathsf{L}}$, is defined inductively as follows:

$$
\begin{array}{lll}
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \top & & \\
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} p & \text{iff} & p \in \sigma(i) \\
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \neg\varphi & \text{iff} & \langle \sigma, i \rangle \not\models^{\mathsf{F}}_{\mathsf{L}} \varphi \\
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi \vee \psi & \text{iff} & \langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi \text{ or } \langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \psi \\
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \mathsf{X}\varphi & \text{iff} & i + 1 < |\sigma| \text{ and } \langle \sigma, i+1 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi \\
\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi \,\mathsf{U}\, \psi & \text{iff} & \text{there exists } k \in [i, |\sigma|) \text{ such that } \langle \sigma, k \rangle \models^{\mathsf{F}}_{\mathsf{L}} \psi \\
 & & \text{and for all } j \in [i, k),\ \langle \sigma, j \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi
\end{array}
$$

Note in this semantics that in the last position of σ, i.e. when $i = |\sigma| - 1$, the formula $\mathsf{X}\varphi$ does not hold, even for $\varphi = \top$. In fact, it is easy to see that $\langle \sigma, i \rangle \models \neg\mathsf{X}\top$ if and only if $i = |\sigma| - 1$.

Based on this semantics, we can formalize what it means for a partial order trace to satisfy a LTL formula. Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a LTL formula φ, we say that T satisfies φ, noted $T \models_{\mathsf{L}} \varphi$ if and only if $\mathsf{traces}(K_T) \subseteq [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$, where $[\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}} \stackrel{\mathbf{def}}{=} \{\sigma \in (2^{\mathbb{P}})^+ \mid \langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi\}$. In other words, T satisfies φ if every finite trace of K_T satisfies φ. This leads us to the problem of determining whether a po-trace T satisfies a LTL formula φ. Following the authors of [Jard et al., 1994], we refer to this problem as the *trace checking* problem[2].

Definition 4.7 (LTL Trace Checking Problem)

Given a po-trace T over a set of propositions $\mathbb{P}$ and a LTL formula φ over $\mathbb{P}$, the LTL trace checking problem (LTL-TC) consists in determining if $T \models_{\mathsf{L}} \varphi$.

Since the dual of the predicate detection problem can be solved using LTL, we know that the LTL trace checking problem is at least as hard. In fact, as it turns out, LTL-TC is coNP-complete. We first show, that LTL-TC is in coNP. For that, we examine the problem of determining if a given finite propositional sequence σ satisfies a LTL formula φ. This problem has been studied in [Markey and Schnoebelen, 2003], and more extensively in [Markey, 2003] where the author present a polynomial time algorithm.

[2] the term *trace checking* is also used in a slightly different context in [Augusto et al., 2003]

Lemma 4.3 ([Markey and Schnoebelen, 2003])

Given a set of propositions $\mathbb{P}$, determining if a finite non-empty propositional sequence $\sigma \in (2^{\mathbb{P}})^+$ satisfies a given LTL formula φ, i.e. if $\langle \sigma, 0 \rangle \models \varphi$, is in PTIME.

Proof Sketch

The problem can be solved using dynamic programming, as presented in Algorithm 4.3. This algorithm works as follows. It fills a boolean table $t[i]$ with $i \in [0, |\sigma|)$, in such a way that $t[i]$ is true if and only if $\langle \sigma, i \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi$. This algorithm computes one table of size $|\sigma|$ for each sub-formulae of φ, and therefore has a time complexity in $O(|\sigma| \times |\varphi|)$.

This complexity result can be used in turn to establish an upper bound on the complexity of LTL-TC.

Lemma 4.4

The trace checking problem for LTL is in coNP.

Proof

We show equivalently that the dual problem of determining if $T \not\models_{\mathsf{L}} \varphi$ is in NP. A non-deterministic algorithm only has to guess finite traces of K_T. This can be done in polynomial time by executing an arbitrary sequence of events compatible with $\preceq$ and compute the valuation at each step. Then, for each of those traces σ, the algorithm checks if $\langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \neg\varphi$, which can be done in polynomial time according to Lemma 4.3. The algorithm return true if and only if a trace σ satisfying $\neg\varphi$ is found. Indeed, by Definition 4.6, $\langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \neg\varphi$ holds if and only if $\langle \sigma, 0 \rangle \not\models^{\mathsf{F}}_{\mathsf{L}} \varphi$. Therefore such a run exists if and only if $\mathsf{traces}(K_T) \not\subseteq [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$. Hence, we conclude.

Moreover, a direct consequence of Lemma 4.3 is that LTL-TC can be solved in polynomial time for total order traces, since there is only one run to check.

Theorem 4.4 (Complexity of LTL-TC for Total Order Traces)

The trace checking problem for LTL is in PTIME for total order trace.

Proof

A total order trace T admits exactly one finite trace σ, since its set of events is totally ordered. Determining if $T \models_{\mathsf{L}} \varphi$ can then be done by checking that $\langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi$, which can be done in polynomial time according to Lemma 4.3.

We now turn our attention to the lower bound on the complexity of LTL-TC. For that, once again, we consider the dual problem, and prove its NP-hardness.

```
1  function satLTL(σ, φ)
   input  : a finite sequence σ ∈ (2^ℙ)^+ and a LTL formula φ
   returns: a boolean table t such that ∀i ∈ [0, |σ|), t[i] is true iff ⟨σ, i⟩ ⊨^F_L φ
2  begin
3      if φ = ⊤ then
4          for i := 0 to |σ| − 1 do  t[i] := tt
5      else if φ = p then
6          for i := 0 to |σ| − 1 do  t[i] := (p ∈ σ(i))
7      else if φ = ¬φ1 then
8          t1 := satLTL(σ, φ1)
9          for i := 0 to |σ| − 1 do  t[i] := ¬t1[i]
10     else if φ = X φ1 then
11         t1 := satLTL(σ, φ1), t[|σ| − 1] := ff
12         for i := 0 to |σ| − 2 do  t[i] := t1[i + 1]
13     else if φ = φ1 ∨ φ2 then
14         t1 := satLTL(σ, φ1), t2 := satLTL(σ, φ2)
15         for i := 0 to |σ| − 1 do  t[i] := (t1[i] ∨ t2[i])
16     else if φ = φ1 U φ2 then
17         t1 := satLTL(σ, φ1), t2 := satLTL(σ, φ2)
18         t[|σ| − 1] := t2[|σ| − 1]
19         for i := |σ| − 2 downto 0 do
20             t[i] := (t2[i] ∨ (t[i + 1] ∧ t2[i]))
21     returntt;
22 end
```

Algorithm 4.3 - Satisfaction of LTL formulae over finite sequences [Markey, 2003]

Lemma 4.5

The trace checking problem for LTL is coNP-hard.

Proof

We show equivalently that the dual problem of determining if $T \not\models_{\mathsf{L}} \varphi$ is NP-hard. For that purpose, we reduce PRED. Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a PBL formula ψ, we show that $T \not\models_{\mathsf{L}} \mathsf{G}\neg\psi$ if and only if $T \models_{\mathsf{P}} \psi$.

$\boxed{\Rightarrow}$ If $T \not\models_{\mathsf{L}} \mathsf{G}\neg\psi$, it means that there exists a trace $\sigma \in \mathsf{traces}(K_T)$ such that $\langle \sigma, 0 \rangle \not\models^{\mathsf{F}}_{\mathsf{L}} \mathsf{G}\neg\psi$, or equivalently that $\langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \neg\mathsf{G}\neg\psi = \mathsf{F}\,\psi$. It follows by Definition 4.6, that there exists $k \in [0, |\sigma|)$ such that $\langle \sigma, k \rangle \models^{\mathsf{F}}_{\mathsf{L}} \psi$. However, since ψ is a PBL formula, we have that $\langle \sigma, k \rangle \models^{\mathsf{F}}_{\mathsf{L}} \psi$ if and only if $\sigma(k) \models_{\mathsf{P}} \psi$. Then, let $\rho \in \mathsf{runs}(K_T)$ be the run corresponding to σ, i.e. $\mathcal{L}(\rho) = \sigma$. Since $\rho(k) \in Q$ and $\mathcal{L}(\rho(k)) = \sigma(k)$, we can conclude that $\exists C \in Q : \mathcal{L}(C) \models_{\mathsf{P}} \psi$ and therefore that $T \models_{\mathsf{P}} \psi$.

$\boxed{\Leftarrow}$ If $T \models_{\mathsf{P}} \psi$, we have that $\exists C \in Q : \mathcal{L}(C) \models_{\mathsf{P}} \psi$. Then, since $\varnothing \subseteq C \subseteq E$, by Lemma 4.2, we have that $\varnothing \rightsquigarrow C \rightsquigarrow E$. We can therefore conclude that there exists a finite trace $\sigma \in \mathsf{traces}(K_T)$ such that $\mathcal{L}(C)$ occurs in σ, i.e. $\exists k \in [0, |\sigma|) : \sigma(k) = \mathcal{L}(C)$. By Definition 4.6, it follows directly that $\langle \sigma, 0 \rangle \models^{\mathsf{F}}_{\mathsf{L}} \mathsf{F}\,\psi = \neg\mathsf{G}\neg\psi$, or equivalently that $\langle \sigma, 0 \rangle \not\models^{\mathsf{F}}_{\mathsf{L}} \mathsf{G}\neg\psi$. Hence, we conclude.

As a direct consequence of Lemma 4.4 and Lemma 4.5, the LTL trace checking problem is coNP-complete.

Theorem 4.5 (Complexity of LTL-TC)

The trace checking problem for LTL is coNP-complete.

4.3.3 CTL Properties

An alternative approach to specifying temporal properties is to use the branching time temporal logic CTL instead. This approach also generalizes the predicate detection problem. Indeed, reachability questions can easily be encoded in CTL using the EF modality. More precisely, given a po-trace T and a PBL formula φ, we could equivalently ask if $\mathsf{EF}\,\varphi$ is satisfied by T, i.e. if there exists a run of T in which φ eventually holds, which is exactly the same thing as asking if T contains a cut in which φ holds. However, similarly to what we faced for LTL, we need to adapt the traditional semantics of CTL

in order to take into account the finite nature of partial order traces. In this case, we directly define the semantics over partial order traces. The formal definition follows.

Definition 4.8 (Semantics of CTL over Partially Ordered Traces)

Given a set of propositions $\mathbb{P}$, let φ, ψ be two CTL formulae over $\mathbb{P}$, $p \in \mathbb{P}$ be a proposition, $T = \langle E, V_0, \delta, \preceq \rangle$ be a partial order trace such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and $C \in Q$ be a cut of T. The satisfaction relation for CTL over partial order traces, noted $\models^{\mathsf{T}}_{\mathsf{C}}$, is defined inductively as follows:

$$
\begin{array}{lll}
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \top & & \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \neg\varphi & \text{iff} & \langle T, C \rangle \not\models^{\mathsf{T}}_{\mathsf{C}} \varphi \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} p & \text{iff} & p \in \mathcal{L}(C) \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi \vee \psi & \text{iff} & \langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi \text{ or } \langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}\,\varphi & \text{iff} & \text{there exists } D \in \mathsf{post}(C) \text{ such that } \langle T, D \rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}\,\varphi & \text{iff} & \text{for all } D \in \mathsf{post}(C),\ \langle T, D \rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{E}(\varphi\,\mathsf{U}\,\psi) & \text{iff} & \text{there exists } \rho \in \mathsf{runs}(C) \text{ and } k \in [0, |\rho|) \text{ such that} \\
 & & \langle T, \rho(k) \rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi \text{ and for all } j \in [0, k),\ \langle T, \rho(j) \rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi \\
\langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{A}(\varphi\,\mathsf{U}\,\psi) & \text{iff} & \text{for all } \rho \in \mathsf{runs}(C), \text{ there exists } k \in [0, |\rho|) \text{ such that} \\
 & & \langle T, \rho(k) \rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi \text{ and for all } j \in [0, k),\ \langle T, \rho(j) \rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi
\end{array}
$$

Based on this semantics, we can easily formalize the CTL trace checking problem. Given a trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a CTL formula φ, we say that a po-trace T satisfies a CTL formula φ, noted $T \models_{\mathsf{C}} \varphi$, if and only if $\varnothing \in [\![\varphi]\!]^T_{\mathsf{C}}$, where $[\![\varphi]\!]^T_{\mathsf{C}} \stackrel{\mathbf{def}}{=} \{C \in Q \mid \langle T, C \rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi\}$. In other words, T satisfies φ if the formula is satisfied in the initial cut of T, i.e. the empty cut. The trace checking problem for CTL (CTL-TC) can then be formalized as follows.

Definition 4.9 (CTL Trace Checking Problem)

Given a partial order trace T over a set of propositions $\mathbb{P}$, and a CTL formula φ over $\mathbb{P}$, the CTL trace checking problem (CTL-TC) consists in determining if $T \models_{\mathsf{C}} \varphi$.

As we have seen previously, the predicate detection problem can be encoded in CTL. Therefore, the CTL trace checking problem is at least as hard. In fact, it is even harder. Indeed, as it turns our, CTL-TC is PSPACE-complete. We first show the complexity upper bound.

Lemma 4.6

The computational tree logic trace checking problem is in PSPACE.

Proof

We exhibit an algorithm that recursively determines, given a cut C of T and a CTL formula φ, if $\langle T, C\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi$. The algorithm works as follows. If $\varphi = \top$, the algorithm directly answers positively. If $\varphi = p$, the algorithm returns true if and only if $p \in \mathcal{L}(C)$. This can be done in polynomial time, hence in polynomial space, by examining the events of C. If $\varphi = \neg\varphi_1$ or $\varphi = \varphi_1 \vee \varphi_2$, the algorithm first recursively determines whether the subformulae of φ hold in C, and then use the semantics, presented in Definition 4.8, to answer. If $\varphi = \mathsf{EX}\,\varphi_1$ or $\varphi = \mathsf{AX}\,\varphi_1$, the algorithm recursively determines in each successor of C if the formula φ_1 holds, and then answers according to the semantics. If $\varphi = \mathsf{E}(\varphi_1\,\mathsf{U}\,\varphi_2)$, the algorithm answers by exploiting the fact that φ is equivalent to $\varphi_2 \vee (\varphi_1 \wedge \mathsf{EX}\varphi)$. If $\varphi = \mathsf{A}(\varphi_1\,\mathsf{U}\,\varphi_2)$, the algorithm answers by exploiting the fact that φ is equivalent to $\varphi_2 \vee (\varphi_1 \wedge \mathsf{AX}\varphi)$ if $C \neq E$, and to φ_2 otherwise. In this algorithm, each context contains only one cut and one formula. The size of each context is therefore bounded by $|E| + |\varphi|$. Moreover, at each recursive call, either the size of C strictly increases or the size of φ strictly decreases. Since $|C| \leq |E|$ and $|\varphi| \geq 1$, the algorithm will need at most $(|E \setminus C| + 1) \times |\varphi|$ nested recursive calls. We can therefore conclude that the algorithm can determine if $\langle T, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi$ in a space bounded by $\big((|E| + 1) \times |\varphi|\,\big) \times \big(\,|E| + |\varphi|\,\big)$, thus establishing the result.

Then, we turn our attention to PSPACE-hardness. In order to prove that, we reduce QBL-SAT. Given a fully quantified QBL formula φ, we build a partial order trace $T_\varphi = \langle E, V_0, \delta, \preceq\rangle$ over a set of new propositions $\{q_{p,\mathsf{tt}}, q_{p,\mathsf{ff}} \mid p \in \mathsf{prop}(\varphi)\}$ and a CTL formula ψ_φ over this set such that $\exists V \subseteq \mathbb{P} : V \models_{\mathsf{Q}} \varphi$ if and only if $T_\varphi \models_{\mathsf{C}} \psi_\varphi$. The partial order trace is build as follows: (i) the set of events is composed of two events for each propositions p appearing in φ, i.e. $E \stackrel{\mathbf{def}}{=} \{e_{p,b} \mid p \in \mathsf{prop}(\varphi) \wedge b \in \mathbb{B}\}$, (ii) none of the propositions are initially satisfied, i.e. $V_0 = \varnothing$, (iii) each event $e_{p,b}$ sets the corresponding propositions $q_{p,b}$ to tt, i.e. $\forall e_{p,b} \in E : (\delta(e_{p,b}, \mathsf{tt}) = \{q_{p,b}\}) \wedge (\delta(e_{p,b}, \mathsf{ff}) = \varnothing)$, and finally (iv) none of the events are ordered, i.e. $\preceq \stackrel{\mathbf{def}}{=} \varnothing$. The CTL formula ψ_φ is defined inductively as follows:

$$\psi_\varphi \stackrel{\mathbf{def}}{=} \begin{cases} \varphi[p/\mathsf{value}(p,\mathsf{tt})] & \text{if } \varphi \text{ is a PBL formula} \\ \mathsf{EX}\big(\,(\mathsf{value}(p,\mathsf{tt}) \vee \mathsf{value}(p,\mathsf{ff})) \wedge \psi_{\varphi_1}\,\big) & \text{if } \varphi = \exists p\,\varphi_1 \\ \mathsf{AX}\big(\,(\mathsf{value}(p,\mathsf{tt}) \vee \mathsf{value}(p,\mathsf{ff})) \Rightarrow \psi_{\varphi_1}\,\big) & \text{if } \varphi = \forall p\,\varphi_1 \end{cases}$$

where $\mathsf{value}(p,b) \stackrel{\mathbf{def}}{=} (q_{p,b} \wedge \neg q_{p,\bar{b}})$.

The intuition behind this reduction is the following. Decisions about the truth value of the propositions appearing in φ are encoded in the cuts of the po-trace. If a cut C contains no event about p, i.e. $\{e_{p,\mathsf{tt}}, e_{p,\mathsf{ff}}\} \not\subseteq C$, it means that the truth value of p has not been decided yet. On the other hand, if C contains an event $e_{p,b}$, it means that the boolean value b has been assigned to p. If C contains both $e_{p,\mathsf{tt}}$ and $e_{p,\mathsf{ff}}$, it means that contradictory decisions have been taken about p. With this encoding, starting from the initial cut $\varnothing$, where no decisions have been taken (since $V_0 = \varnothing$), we can model the decision of assigning a boolean value b to a proposition p, by triggering the corresponding event $e_{p,b}$. The CTL formula is then used to encode the order in which the decisions have to be taken, i.e. the order dictated by the quantifiers appearing in φ. For that purpose, existential, respectively universal, quantifiers are modelled using EX, respectively AX.

Lemma 4.7

A fully quantified QBL formula φ is satisfiable if and only if $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_\varphi$.

Proof

We proceed by induction on $|\mathsf{prop}(\varphi)|$.

Initial Step If $|\mathsf{prop}(\varphi)| = 0$, we have that φ is a boolean combination of $\top$'s, i.e. φ is either equivalent to $\top$ or $\bot$. In both cases, by construction, we have that $\psi_\varphi = \varphi$. It follows immediately that φ is satisfiable if and only if $\langle T_\varphi, \varnothing\rangle \not\models^{\mathsf{T}}_{\mathsf{C}} \psi_\varphi$.

Induction Step We have to consider two cases:

(i) The first case to consider is when $\varphi = \exists p\, \varphi_1$. In this case, since φ is fully quantified, by Definition 1.4, we have that φ is satisfiable if and only if $\varphi_1[p/\top]$ or $\varphi_1[p/\bot]$ is satisfiable. By induction, this holds if and only if $\langle T_{\varphi_1[p/\top]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1[p/\top]}$ or $\langle T_{\varphi_1[p/\bot]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1[p/\bot]}$. By construction of ψ_φ, this is equivalent to $\langle T_{\varphi_1[p/\top]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}[q_{p,\mathsf{tt}}/\top, q_{p,\mathsf{ff}}/\bot]$ or $\langle T_{\varphi_1[p/\bot]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}[q_{p,\mathsf{tt}}/\bot, q_{p,\mathsf{ff}}/\top]$ and by construction of T_φ to $\langle T_\varphi, \{e_{p,\mathsf{tt}}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$ or $\langle T_\varphi, \{e_{p,\mathsf{ff}}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$. Then, for any $b \in \mathbb{B}$, since $e_{p,b}$ is the only event capable of satisfying $\mathsf{value}(p, b)$, we can deduce that $\langle T_\varphi, \{e_{p,b}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$ holds if and only if $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}(\mathsf{value}(p, b) \wedge \psi_{\varphi_1})$. Therefore, φ is satisfiable if and only if $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}(\mathsf{value}(p, \mathsf{tt}) \wedge \psi_{\varphi_1})$ or $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}(\mathsf{value}(p, \mathsf{ff}) \wedge \psi_{\varphi_1})$. By Definition 4.8, this is equivalent to $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}(\mathsf{value}(p, \mathsf{tt}) \wedge \psi_{\varphi_1}) \vee \mathsf{EX}(\mathsf{value}(p, \mathsf{ff}) \wedge \psi_{\varphi_1})$, and again by Definition 4.8, to $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{EX}\big((\mathsf{value}(p, \mathsf{tt}) \vee \mathsf{value}(p, \mathsf{ff})) \wedge \psi_{\varphi_1}\big) \stackrel{\mathbf{def}}{=} \psi_\varphi$.

Proof (cont'd)

(*ii*) The second case to consider is when $\varphi = \forall p\, \varphi_1$. In this case, since φ is fully quantified, by Definition 1.4, we have that φ is satisfiable if and only if $\varphi_1[p/\top]$ and $\varphi_1[p/\bot]$ is satisfiable. By induction, this holds if and only if $\langle T_{\varphi_1[p/\top]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1[p/\top]}$ and $\langle T_{\varphi_1[p/\bot]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1[p/\bot]}$. By construction of ψ_φ, this is equivalent to $\langle T_{\varphi_1[p/\top]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}[q_{p,\mathsf{tt}}/\top, q_{p,\mathsf{ff}}/\bot]$ and $\langle T_{\varphi_1[p/\bot]}, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}[q_{p,\mathsf{tt}}/\bot, q_{p,\mathsf{ff}}/\top]$ and by construction of T_φ to $\langle T_\varphi, \{e_{p,\mathsf{tt}}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$ and $\langle T_\varphi, \{e_{p,\mathsf{ff}}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$. Then, for any cut $C \in \mathsf{post}(\varnothing)$, we have that either $C = \{e_{p,b}\}$ for a certain $b \in \mathbb{B}$, in which case $\langle T_\varphi, C\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{value}(p,b)$, otherwise $\langle T_\varphi, C\rangle \not\models^{\mathsf{T}}_{\mathsf{C}} \mathsf{value}(p,b)$ for all $b \in \mathbb{B}$. It follows that for any $b \in \mathbb{B}$, $\langle T_\varphi, \{e_{p,b}\}\rangle \models^{\mathsf{T}}_{\mathsf{C}} \psi_{\varphi_1}$ holds if and only if $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}(\mathsf{value}(p,b) \Rightarrow \psi_{\varphi_1})$. Therefore, φ is satisfiable if and only if $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}(\mathsf{value}(p,\mathsf{tt}) \Rightarrow \psi_{\varphi_1})$ or $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}(\mathsf{value}(p,\mathsf{ff}) \Rightarrow \psi_{\varphi_1})$. By Definition 4.8, this is equivalent to $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}(\mathsf{value}(p,\mathsf{tt}) \Rightarrow \psi_{\varphi_1}) \wedge \mathsf{AX}(\mathsf{value}(p,\mathsf{ff}) \Rightarrow \psi_{\varphi_1})$, and again by Definition 4.8, to $\langle T_\varphi, \varnothing\rangle \models^{\mathsf{T}}_{\mathsf{C}} \mathsf{AX}\big((\mathsf{value}(p,\mathsf{tt}) \vee \mathsf{value}(p,\mathsf{ff})) \Rightarrow \psi_{\varphi_1}\big) \stackrel{\mathbf{def}}{=} \psi_\varphi$.

It follows directly that the CTL trace checking problem is PSPACE-complete.

Theorem 4.6 (Complexity of CTL-TC)

The computational tree logic trace checking problem is PSPACE-complete.

Finally, let us also examine the theoretical complexity in the case of total order traces. In this case, it turns out that the trace checking problem can be solved in polynomial time.

Theorem 4.7 (Complexity of CTL-TC for Total Order Traces)

The trace checking problem for CTL is in PTIME for total order traces.

Proof

On total order traces, for every $C \in Q$, we have that $\mathsf{post}(C)$ is a singleton. Therefore, the semantics of EX and AX coincide with that of the X operator in LTL. Moreover, for every $C \in Q$, $\mathsf{runs}(C)$ contains exactly one run. Therefore the semantics of EU and AU coincide with that of the U operator in LTL. One can therefore build, in polynomial time, a LTL formula ψ from φ, by replacing EX and AX by X, and EU and AU by U, and check equivalently that $T \models_{\mathsf{L}} \psi$. By Theorem 4.4, this can be done in polynomial time.

Problem	Total Order Traces	Partial Order Traces
PRED	in PTIME(Theorem 4.3)	NP-complete (Theorem 4.2)
LTL-TC	in PTIME(Theorem 4.4)	coNP-complete (Theorem 4.5)
CTL-TC	in PTIME(Theorem 4.7)	PSPACE-complete (Theorem 4.6)

Table 4.1 - Summary of complexity results on partial and total order traces

4.4 Discussions

In this chapter, we have examined in detail the first two steps toward testing distributed systems, namely instrumentation and specification. For the instrumentation step, we have presented techniques that allow, both for message-passing and multithreaded programs, to obtain a partial order on the events emitted by the system. We also introduced a model to capture this partial order, namely *partial order traces*, in which events are abstracted as predicate transformers. The main advantage of this model is that it is very general. For instance, it completely abstracts away the communication scheme used by the system.

In this framework, we studied the predicate detection problem, i.e. determining if a partial order trace contains a cut satisfying a given predicate expressed as a PBL formula. We also studied the trace checking problem, i.e. determining if a po-trace satisfies a temporal property expressed either in LTL or CTL. The theoretical complexities of those problems are summarized in Table 4.1. As we can see in this table, in every instance, the problem becomes much simpler when a total order on the events of the system is assumed. This motivates the study of partial order traces. Another interesting observation is that, contrarily to model checking, in the case of trace checking, LTL is less expensive that CTL.

In the remaining three chapters of this dissertation, we tackle the last step of the testing process, i.e. trace analysis. More precisely, we will examine in detail how the predicate detection problem, and the trace checking problem (respectively for LTL and CTL) can be solved in practice. We will present existing solutions, and when possible build on those to come up with more efficient and practical algorithms.

Chapter

5

Predicate Detection

« *Once you have eliminated the impossible, whatever remains, however improbable, must be the truth* »

Sherlock Holmes, in *The Sign of the Four*

PREDICATE detection consists in determining if a distributed execution admits a global state where a certain property holds. In the previous chapter, we have seen how such a distributed execution can be modeled as a po-trace, where global states are modeled as cuts. We also saw how properties on those po-traces can be specified using formal logics. In this chapter, we concentrate on properties expressed as PBL formulae. Determining if a po-trace satisfies such a property can then be formalized as follows. Given a po-trace T and a PBL formula φ, does there exist a cut C of T such that φ holds in C. A straightforward solution to this problem is therefore to explore the lattice of cuts of T, and for each cut in this lattice check whether the formula is satisfied or not. However, as we have seen in Chapter 4, the number of cuts in this lattice can be exponentially bigger than the number of events. Because of this, exploring the entire lattice of cuts is not practical. Unfortunately, we also saw that the predicate detection problem is NP-complete, which makes this exponential blowup unavoidable. Nevertheless, throughout the literature, several restricted classes of predicates have been studied for which efficient, i.e. polynomial, detection algorithms have been developed. However, these classes of predicates have a semantical definition. It is therefore not, in general, easy to check whether a given PBL formula belongs to one of those classes. For that reason, we study the problem from a syntactical point of view, and come up with a notable result: the predicate detection problem is easy for formulae in disjunctive normal form.

The remainder of this chapter is structured as follows. First, in Section 5.1, we review some of the known classes of predicates, and present the corresponding detection algorithms adapted to our framework. Then, in Section 5.2, we give syntactical sufficient conditions for two of the most important classes of predicates, leading us to show that DNF predicates are easy to detect. Finally, we conclude, in Section 5.3, by giving a brief summary of all the results of this chapter.

5.1 Classes of Predicates

As already mentioned in the introduction, a simple way to determine if a po-trace contains a cut where a given PBL formula φ holds can be done by exploring the lattice of cuts of the trace and, for each cut C, check whether φ holds in C or not. This technique is formalized in Algorithm 5.1. This algorithm works as follows. It maintains two sets of cuts. The first set W is used to keep track of the cuts that remain to be explored, while the second set Z is used to remember the cuts that have already been explored (line 11). This allows to avoid exploring a cut more than once. The algorithm starts with the initial empty cut in W (line 11). Then, while W is non-empty (line 12), the algorithm picks a cut C from W, add it to Z (line 13), and checks whether $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$ (line 15) using the function $\mathsf{models}(T, C, \varphi)$. If it is the case, the algorithm returns true (line 16). Otherwise, the algorithm adds to W all successors of C that are not already in Z (lines 17–18). If the main loop terminates without finding a satisfying cut, the algorithm returns false. In this algorithm, each cut of the lattice is examined at most once. However, the number of cuts in this lattice can be exponentially bigger than the number of events. For each of those cuts, the algorithm checks the truth value of φ. For that purpose, the function models first computes in a variable V, the valuation corresponding to $\mathcal{L}(C)$ (lines 3–6). Assuming an appropriate representation for cuts, i.e. one that allows to obtain $\mathsf{max}_{\preceq}(C_{/p})$ in $O(1)$, this can be one in $O(|\varphi|)$, since $|\mathsf{prop}(\varphi)| \leq |\varphi|$. Then, the function returns tt if and only if $V \models_{\mathsf{P}} \varphi$. This can also be done in $O(|\varphi|)$, by recursively applying Definition 1.2. The main algorithm also computes the set of enabled events for each cut. In order to do this, the algorithm only has to examine the next event of each of the k processes. This can therefore be done in $O(k)$. Hence, the overall time complexity of Algorithm 5.1 is in $O(2^{|E|} \times |\varphi| \times k)$.

In an effort to address this exponential blowup, many classes of restricted predicates have been defined throughout the literature which are easier to detect, e.g. *local*, *disjunctive*, *conjunctive*, *stable*, *observer-independent*, *linear*, and *regular* predicates. Those classes of predicates are reviewed in Section 5.1.1 to Section 5.1.7.

```
1  function models(T, C, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a cut C, a predicate φ
   returns: tt if and only if L(C) ⊨_P φ
2  begin
3      V := V_0
4      forall p ∈ prop(φ) s.t. C_/p ≠ ∅ do
5          e := max_⪯(C_/p)
6          V := (V \ δ(e, ff)) ∪ δ(e, tt)
7      return V ⊨_P φ
8  end

9  function detectArbitrary(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, an arbitrary predicate φ
   returns: tt if and only if T ⊨_P φ
10 begin
11     W := {∅}, Z = ∅
12     while W ≠ ∅ do
13         Choose C ∈ W
14         W := W \ {C}, Z := Z ∪ {C}
15         if models(T, C, φ) then
16             returntt
17         forall e ∈ enabled(C) do
18             W := (W ∪ {C ∪ {e}}) \ Z
19     returnff
20 end
```

Algorithm 5.1 - Detection of arbitrary predicates

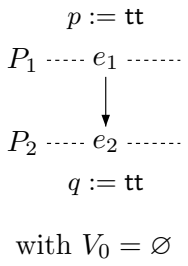


Figure 5.1 - A po-trace where $p \wedge q$ is local to P_2

5.1.1 Local Predicates

The first class of predicates we examine is that of *local* predicates, i.e. predicates for which the truth value only depends on the local states of one process. The formal definition, taken from [Charron-Bost et al., 1995], follows.

Definition 5.1 (Local Predicate [Charron-Bost et al., 1995])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is local to a process P_i of T if and only if

$$\forall C, D \in Q : \big(C \cap P_i = D \cap P_i\big) \Rightarrow \big((\mathcal{L}(C) \models_{\mathsf{P}} \varphi) \Leftrightarrow (\mathcal{L}(D) \models_{\mathsf{P}} \varphi)\big)$$

As illustrated by Example 5.1, if the propositions appearing in a predicate are only modified by one process P_i, then this predicate is local to P_i.

Example 5.1

Consider the po-trace of Figure 4.4(a) (page 95). The predicate $\varphi_1 \stackrel{\text{def}}{=} \neg r$ is local to process P_1 and the predicate $\varphi_2 \stackrel{\text{def}}{=} s$ is local to process P_2.

However, as illustrated in Example 5.2, this not a necessary condition.

Example 5.2

Consider the po-trace of Figure 5.1 with $V_0 = \varnothing$. We have that the predicate $\varphi = p \wedge q$ is local process P_2. Indeed, before event e_2 is triggered, i.e. in the cuts $\varnothing$ and $\{e_1\}$, the predicate is false. On the other hand, after e_2 is triggered, i.e. in the cut $\{e_1, e_2\}$, the predicate is true.

In that sense, the denomination *local*, as formalized in [Charron-Bost et al., 1995], can be a little misleading. Indeed, the truth value of a predicate φ local to some process P_i does not depend only on the events of P_i, but might also depend on events from other processes.

```
1 function detectLocal(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a predicate φ local to process P_i
  returns: tt if and only if T ⊨_P φ
2 begin
3 |  S_i := ∅
4 |  while (P_i \ S_i ≠ ∅) ∧ (¬models(T, ↓S_i, φ)) do
5 |  |  S_i := S_i ∪ min_⪯(P_i \ S_i)
6 |  return models(T, ↓S_i, φ)
7 end
```

Algorithm 5.2 - Detection of local predicates

By definition, detecting local predicates can be done by examining one cut for each local state $S_i \subseteq P_i$, like e.g. $\downarrow S_i$. Therefore, as formalized in Algorithm 5.2, starting from the initial empty local state (line 3), we can successively enumerate the local states of P_i until a satisfying cut is found (lines 4–5). In this algorithm, at most $|P_i| + 1$ cuts are examined, and for each of those cuts, the truth value of φ is checked, which can be done in $O(|\varphi|)$, as shown previously, assuming of course that $\downarrow S_i$ can be computed in $O(1)$. Hence the overall time complexity of Algorithm 5.2 is in $O(|P_i| \times |\varphi|)$.

5.1.2 Disjunctive Predicates

The second class of predicates we examine is that of *disjunctive* predicates which are disjunctions of local predicates. The formal definition follows.

Definition 5.2 (Disjunctive Predicate)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes, a disjunctive predicate is a predicate of the form $\varphi_1 \vee \ldots \vee \varphi_k$ where, for any $i \in [1, k]$, φ_i is a predicate local to process P_i.

In the previous definition, for simplicity, we assume that disjunctive predicates have exactly one branch φ_i for each process P_i. This assumption can be made without loss of generality. Indeed, if a local predicate is missing for a certain process P_i, one can always chose $\varphi_i = \bot$, which is trivially local to any process. Conversely, if for some process P_i, there are more than one local predicates $\varphi_{i,1}, \varphi_{i,2}, \ldots, \varphi_{i,\ell}$, one can always group them together and choose $\varphi_i = \varphi_{i,1} \vee \varphi_{i,2} \vee \ldots \vee \varphi_{i,\ell}$, which by Definition 5.1 is also local to process P_i.

```
1 function detectDisjunctive(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a disjunctive predicate φ_1 ∨ ... ∨ φ_k
  returns: tt if and only if T ⊨_P φ_1 ∨ ... ∨ φ_k
2 begin
3     forall i ∈ [1, k] do
4         if detectLocal(T, φ_i) then
5             return tt
6     returnff
7 end
```

Algorithm 5.3 - Detection of disjunctive predicates

Example 5.3

Consider the po-trace of Figure 4.4(a) (page 95). The predicate $\varphi_1 \stackrel{\text{def}}{=} \neg r \vee s$ is disjunctive, since $\neg r$ is local to process P_1 and s is local to process P_2. Furthermore, the predicate $\varphi_2 \stackrel{\text{def}}{=} \neg r \vee p \vee \bot$ is also disjunctive, since $\neg r \vee p$ is local to process P_1 and $\bot$ is local to process P_2.

The detection of disjunctive predicates can be reduced to the local case. Indeed, each branch of the predicate can be examined independently.

Proposition 5.1

Given a po-trace T, and a disjunctive predicate $\varphi_1 \vee \varphi_2 \vee \ldots \varphi_k$, we have that:

$$(T \models_{\mathrm{P}} \varphi_1 \vee \varphi_2 \vee \ldots \varphi_k) \Leftrightarrow (T \models_{\mathrm{P}} \varphi_1 \vee T \models_{\mathrm{P}} \varphi_1 \vee \ldots \vee T \models_{\mathrm{P}} \varphi_k)$$

Proof

The proof is immediate from Definitions 1.2 and 4.4.

Therefore, one can use Algorithm 5.2 repeatedly for each branch φ_i, and stop as soon as a satisfying branch is found. Using this method, formalized in Algorithm 5.3, at most $\sum_{i\in[1,k]}(|P_i|+1) = |E|+k$ cuts are examined. Algorithm 5.3 therefore has a time complexity in $O(|E| \times |\varphi|)$.

5.1.3 Stable Predicates

The third class of predicates is that of *stable* predicates, first introduced in [Chandy and Lamport, 1985]. Intuitively, stable predicates are predicates that stay true until the end of the execution trace once they become true. Formally, stable predicates are defined as follows.

```
1 function detectStable(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a stable predicate φ
  returns: tt if and only if T ⊨_P φ
2 begin
3 |   return models(T, E, φ)
4 end
```

Algorithm 5.4 - Detection of stable predicate

Definition 5.3 (Stable Predicate [Chandy and Lamport, 1985])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is stable w.r.t. T if and only if

$$\forall C, D \in Q : \big((C \rightsquigarrow D) \wedge (\mathcal{L}(C) \models_{\mathsf{P}} \varphi)\big) \Rightarrow \big(\mathcal{L}(D) \models_{\mathsf{P}} \varphi\big)$$

Inherently, stable predicates are very easy to detect. Indeed, in order to determine if there exists a cut in the trace where φ holds, we only have to examine the last cut of the trace, i.e. the cut containing all the events of T.

Proposition 5.2

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a predicate φ stable w.r.t. T, we have that:

$$(T \models_{\mathsf{P}} \varphi) \Leftrightarrow (\mathcal{L}(E) \models_{\mathsf{P}} \varphi)$$

Proof

Assume $T \models_{\mathsf{P}} \varphi$, or equivalently that $\exists C \in Q : \mathcal{L}(C) \models_{\mathsf{P}} \varphi$. Since $C \in Q = \mathsf{DC}_{\preceq}(E)$, we have that $C \subseteq E \in Q$, which implies by Lemma 4.2, that $C \rightsquigarrow E$. It follows by Definition 5.3 that $\mathcal{L}(E) \models_{\mathsf{P}} \varphi$. Now, assume conversely that $\mathcal{L}(E) \models_{\mathsf{P}} \varphi$, we can immediately conclude that $\exists C \in Q : \mathcal{L}(C) \models_{\mathsf{P}} \varphi$, or equivalently that $T \models_{\mathsf{P}} \varphi$.

This necessary and sufficient condition yields a trivial polynomial time detection method, as presented in Algorithm 5.4. This algorithm simply checks whether φ holds in the last cut of the trace, namely E, which can be done in $O(|\varphi|)$.

Example 5.4

Consider the po-trace of Figure 4.4(a). The predicate $\varphi \stackrel{\text{def}}{=} \neg p$ is stable. Indeed, proposition p is true in the initial valuation, and only event $e_{1,1}$ "modifies" its truth value, by setting it to false. Hence, once e_1 is triggered, φ remains false.

Before moving on to the next class of predicates, let us mention that the class of stable predicates is closed under disjunction and conjunction.

Proposition 5.3 ([Chase and Garg, 1998])

Given a po-trace T, and two predicates φ and ψ stable w.r.t. T, we have that $\varphi \wedge \psi$ and $\varphi \vee \psi$ are both stable w.r.t T.

5.1.4 Observer-Independent Predicates

The next class of predicates is that of *observer-independent* predicates, first introduced in [Charron-Bost et al., 1995]. Observer-independent predicates are predicates such that if they hold for one observation of the po-trace, then they also hold for any other observation. In our framework, observer-independent predicates are defined as follows.

Definition 5.4 (Observer-Independent Predicate [Charron-Bost et al., 1995])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is observer-independent w.r.t. T if and only if

$$\big(\exists \rho \in \mathsf{runs}(K_T), \exists i \in [0, |\rho|) : \mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi\big) \Rightarrow \big(\forall \rho \in \mathsf{runs}(K_T), \exists i \in [0, |\rho|) : \mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi\big)$$

The detection of observer-independent predicates is also quite simple. Indeed, for any predicate φ, we know that φ holds in a cut C if and only if it holds along one run of the po-trace. In the case of observer-independent predicates, this implies that the predicate holds along all runs of the po-trace. Therefore, in this particular case, only examining one run is enough to determine if φ holds. This induces a polynomial time detection method, as presented in Algorithm 5.5. This algorithm simply explores one arbitrary run, and stops as soon as a satisfying cut along this run is found (lines 4–6). In this Algorithm, at most $|E|+1$ cuts are examined. For each of those cuts, the algorithm has to check the truth value of φ and to compute the set of enabled events. Hence, the overall time complexity of Algorithm 5.5 is in $O(|E| \times |\varphi| \times k)$.

Example 5.5

Consider the po-trace of Figure 4.4(a). All the predicates that have been presented so far in Example 5.1, Example 5.3, Example 5.4 are observer-independent.

Before moving on to other predicate classes, let us investigate the relationship between this class of predicates and the two previous ones. First, it is easy to determine that observer-independent predicates include stable predicates.

```
1 function detectObersverIndependent(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, an observer-independent predicate φ
  returns: tt if and only if T ⊨_P φ
2 begin
3     C := ∅
4     while (C ≠ E) ∧ (¬models(T, C, φ)) do
5         Choose e ∈ enabled(C)
6         C := C ∪ {e}
7     return models(T, C, φ)
8 end
```

Algorithm 5.5 - Detection of observer-independent predicate

Proposition 5.4

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, any predicate φ stable w.r.t T is also observer-independent w.r.t. T.

Proof

Let us assume that $\exists \rho \in \mathsf{runs}(K_T), \exists i \in [0, |\rho|) : \mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi$. Since $\mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi$, we know that $T \models_{\mathsf{P}} \varphi$ which implies, by Proposition 5.2, that $\mathcal{L}(E) \models_{\mathsf{P}} \varphi$. However, for every run $\rho \in \mathsf{runs}(K_T)$, the last cut of ρ is E. Therefore, we can conclude that $\forall \rho \in \mathsf{runs}(K_T), \exists i \in [0, |\rho|) : \mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi$, since $\mathcal{L}(\rho(|\rho| - 1)) \models_{\mathsf{P}} \varphi$.

Furthermore, as proven in [Charron-Bost et al., 1995], observer-independent predicates also include disjunctive predicates. To establish this results, the authors first observe that local predicates are also observer-independent.

Proposition 5.5 ([Charron-Bost et al., 1995])

Given a po-trace T of k processes, any predicate φ local to process P_i for some $i \in [1, k]$, is also is observer-independent w.r.t. T.

Then, they remark that observer-independent predicates are closed to disjunction.

Proposition 5.6 ([Charron-Bost et al., 1995])

Given a po-trace T and two predicates φ and ψ observer-independent w.r.t. T, we have that $\varphi \vee \psi$ is observer-independent w.r.t T.

It follows directly that disjunctive predicates are also observer-independent.

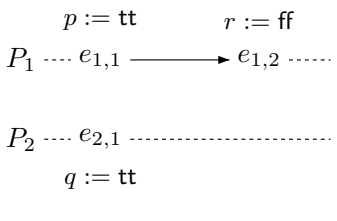


Figure 5.2 - The po-trace T of Example 5.6

Corollary 5.1

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$, any disjunctive predicate φ is also observer-independent w.r.t. T.

Finally, as illustrated in the following example, the classes of stable and disjunctive predicates are incomparable, i.e. none is included in the other.

Example 5.6

Consider the po-trace T of Figure 5.2. The predicate $\varphi_1 \stackrel{\text{def}}{=} p \wedge q$ is stable w.r.t. T, but not disjunctive. Indeed, φ_1 is neither local to P_1, nor to P_2. The predicate $\varphi_2 \stackrel{\text{def}}{=} r$ is disjunctive, since φ_2 is local to process P_1. However, it is not stable, since φ_2 does not hold in the final cut E.

Nevertheless, as illustrated in the following example, those two classes have a non-empty intersection.

Example 5.7

The predicate presented in Example 5.4 is both local to process P_1, and therefore disjunctive, and stable w.r.t. T.

5.1.5 Conjunctive Predicates

The next class of predicates we examine is that of *conjunctive* predicates which are conjunctions of local predicates. Similarly to disjunctive predicates, we assume without loss of generality that conjunctive predicates have exactly one branch for each process. The formal definition follows.

Definition 5.5 (Conjunctive Predicate)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes, a conjunctive predicate is a predicate of the form $\varphi_1 \wedge \ldots \wedge \varphi_k$ where , for any $i \in [1, k]$, φ_i is a predicate local to process P_i.

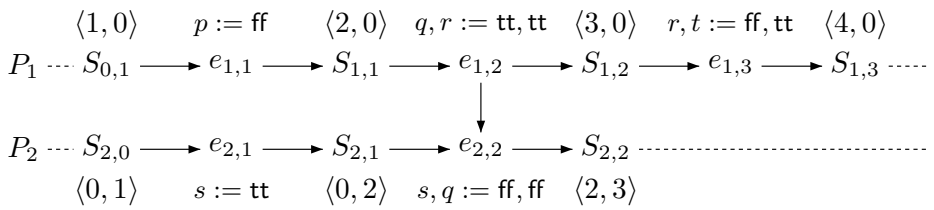


Figure 5.3 - Partial order of the po-trace of Figure 4.4(a) extended to local states

The detection of conjunctive predicates has been studied in [Garg and Waldecker, 1994] where the authors provide a necessary and sufficient condition. In their paper, instead of working on a partial order on events, as can be obtained using the instrumentation technique presented in Section 4.1, the authors use a partial order on the local states of the po-trace. In order to obtain this partial order, the authors define a slightly modified version of the vector clock algorithm, where the vector clock v_i for each process P_i is initialised to 0 for all its components except for $v_i[i]$, which is initialised to 1. The resulting vector clock mapping on local states, noted vc_l in the following, can be easily captured in our framework. Indeed, given a local state $S_i \subseteq P_i$, we have that:

$$vc_l(S_i)[j] = \begin{cases} |{\downarrow}S_i \cap P_j| + 1 & \text{if } i = j \\ |{\downarrow}S_i \cap P_j| & \text{otherwise} \end{cases}$$

In the following, for any two local states S_i, S_j, we note $S_i \preceq_l S_j$ if and only if $vc_l(S_i) \leq vc_l(S_j)$.

Example 5.8

Figure 5.3 shows the partial order of the po-trace of Figure 4.4(a) extended to local states, along with the vector clock mapping vc_l described above. Note, for instance, that local states $S_{1,1}$ and $S_{2,2}$ are ordered, and that we indeed have that $\langle 2, 0\rangle \leq \langle 2, 3\rangle$.

The necessary and sufficient condition can then be formulated as follows.

Proposition 5.7 ([Garg and Waldecker, 1994])

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$, and a conjunctive predicate $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_k$, we have that $T \models_{\mathsf{P}} \varphi$ if and only if

$$\exists S_1 \subseteq P_i, \ldots, S_k \subseteq P_k : \left(\begin{array}{c} \forall i \in [1,k] : \mathcal{L}({\downarrow}S_i) \models_{\mathsf{P}} \varphi_i \\ \wedge \\ \forall i \neq j \in [1,k] : (S_i \not\preceq_l S_j) \wedge (S_j \not\preceq_l S_i) \end{array} \right)$$

```
1  function detectConjunctive(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a conjunctive predicate φ_1 ∧ ... ∧ φ_k
   returns: tt if and only if T ⊨_P φ_1 ∧ ... ∧ φ_k
2  begin
3      forall i ∈ [1, k] do
4          Q[i] := ε, S_i := ∅
5          while S_i ⊆ P_i do
6              if models(T, ↓S_i, φ_i) then
7                  Q[i] := Q[i] · S_i
8              S_i := S_i ∪ min_⪯(P_i \ S_i)
9          if models(T, ↓S_i, φ_i) then
10             Q[i] := Q[i] · S_i
11     return detectAntichainGarg92(Q, ⪯_l)
12 end
```

Algorithm 5.6 - Detection of conjunctive predicate

In other words, there exists a cut satisfying a conjunctive predicate $\varphi_1 \wedge \ldots \wedge \varphi_k$ if and only if there exists an antichain of local states, w.r.t $\preceq_l$, containing exactly one local state S_i for each process P_i such that φ_i holds in S_i. Using this necessary and sufficient condition, the authors of [Garg and Waldecker, 1994] devised a polynomial time algorithm for the detection of conjunctive predicates. This algorithm, formalized in Algorithm 5.6, uses a vector Q of size k, where each entry is a queue of local states. For each process P_i, the algorithm collects in $Q[i]$ all the local states of process P_i where φ_i holds (lines 3–10). This vector of queues is then used to determine if the necessary and sufficient condition is met. For that, the authors of [Garg and Waldecker, 1994], use an antichain detection algorithm from a previous paper [Garg, 1992]. Given a po-set $\langle X, \sqsubseteq \rangle$ partitioned into k chains this detection algorithm return tt if and only if $\langle X, \sqsubseteq \rangle$ admits an antichain of size k. For more details on this algorithm, we refer the reader to [Garg, 1992]. Here, we assume the existence of a function antichainDetectGarg92() (line 11), that takes care of that. In the first part of the algorithm, i.e. the construction of Q, for each process P_i, at most $|P_i| + 1$ local states are considered, and for each of those local states the truth value of φ_i has to be determined. This can be done in $O(|P_i| \times |\varphi_i|)$. Therefore, since $\sum_{i \in [1,k]} |P_i| = |E|$ and $|\varphi_1| \leq |\varphi|$, the time complexity of this part is in $O(|E| \times |\varphi|)$. Then, according to [Garg, 1992], the antichain detection

can be done in $O(k^2 \times m)$, where m is the size of the longest chain in the partial order on local states. Therefore, since $m \leq |E|$ and since $k \leq |\varphi|$, we can conclude that the overall time complexity of Algorithm 5.6 is in $O(|\varphi|^2 \times |E|)$.

Example 5.9

Consider the po-trace of Figure 4.4(a). The predicate $\varphi \stackrel{\text{def}}{=} \neg r \wedge s$ is conjunctive, since $\neg r$ is local to process P_1 and s is local to process P_2. Now consider the partial order on local states of Figure 5.3. We have that $\neg r$ holds in local state $S_{0,1}$, since $V_0 = \varnothing$, and that s hold in local state $S_{2,1}$, since $e_{2,1}$ sets s to true. Therefore, the necessary and sufficient condition of Proposition 5.7 is satisfied, since $\langle 1, 0\rangle$ and $\langle 0, 2\rangle$ are incomparable. It follows directly that φ holds on this po-trace.

5.1.6 Linear Predicates

We now examine the class of *linear* predicates, first introduced in [Chase and Garg, 1995] and further developed in [Chase and Garg, 1998]. This class was also studied in [Garg et al., 2003], where the authors introduced an alternative and equivalent characterization, that we use here. This characterization is based on the notion of *crucial* events. Intuitively, in a cut C, an event e is crucial for a predicate φ if and only if for each cut D reachable from C, D cannot satisfy φ unless it contains e. Formally, the set of events that are crucial in a cut C for a predicate φ is defined as $\mathsf{crucial}(C, \varphi) \stackrel{\text{def}}{=} \{e \in E \mid \forall D \in Q : (C \rightsquigarrow D) \Rightarrow (e \in D \vee \mathcal{L}(D) \not\models_{\mathsf{P}} \varphi)\}$. Based on this notion, a predicate φ is defined as linear if, in every cut $C \neq E$ where φ is not satisfied, there exists at least one enabled event that is crucial for φ. The formal definition follows.

Definition 5.6 (Linear Predicate [Chase and Garg, 1995; Garg et al., 2003])

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$, a predicate φ is linear w.r.t. T if and only if:

$$\forall C \in Q \setminus \{E\} : (\mathcal{L}(C) \not\models_{\mathsf{P}} \varphi) \Rightarrow (\mathsf{enabled}(C) \cap \mathsf{crucial}(C, \varphi) \neq \varnothing)$$

Based on this characterization of linear predicates, the authors of [Garg et al., 2003] propose an algorithm for the detection of linear predicates, which is presented in Algorithm 5.7. This is a simple greedy algorithm that starts with the initial empty cut (line 3). Then, while the predicate is not satisfied (line 4), the algorithm advances through the po-trace by executing an arbitrary enabled crucial event (lines 5–6). The algorithm stops as soon as a satisfying cut is found. Assuming that $\mathsf{crucial}(C, \varphi)$

```
1 function detectLinear(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a linear predicate φ
  returns: tt if and only if T ⊨_P φ
2 begin
3     C := ∅
4     while (C ≠ E) ∧ (¬models(T, C, φ)) do
5         Choose e ∈ enabled(C) ∩ crucial(C, φ)
6         C := C ∪ {e}
7     return models(T, C, φ)
8 end
```

Algorithm 5.7 - Detection of linear predicate

can be computed efficiently[1], i.e. as efficiently as $\mathsf{enabled}(C)$, the time complexity of Algorithm 5.7 is in $O(|E| \times |\varphi| \times k)$. Indeed, in this algorithm, at most $|E|+1$ cuts are considered, and at each step, the truth value of φ and the set of enabled events have to be computed.

Example 5.10

Consider the po-trace of Figure 4.4(a) (page 95). The predicate $\varphi \stackrel{\text{def}}{=} \neg s$ is linear. For instance, in the cut $C \stackrel{\text{def}}{=} \{e_{1,1}, e_{1,2}, e_{2,1}\}$ we have that event $e_{2,2}$ is crucial for φ_1. Indeed, for every cut C such that $C \rightsquigarrow D$, we have that either $e_{2,2} \in D$ or that φ is not satisfied in D. In contrast, the event $e_{1,3}$ is not crucial for φ, since the cut $D \stackrel{\text{def}}{=} \{e_{1,1}, e_{1,2}, e_{2,1}, e_{2,2}\}$ does not contain $e_{1,3}$ and does satisfy φ.

5.1.7 Regular Predicates

Finally, we examine the class of *regular* predicates, first introduced in [Garg and Mittal, 2001b]. Regular predicates are predicates for which the set of satisfying cuts forms a sublattice of the lattice of cuts. In other words, a predicate is regular if and only if it is closed under intersection (the meet operator) and union (the join operator). The formal definition follows.

[1] this assumption is referred to as the *efficient advancement property* in [Garg et al., 2003]

Definition 5.7 (Regular Predicate [Garg and Mittal, 2001b])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is regular w.r.t. T if and only if:

$$\forall C, D \in Q : \big(\mathcal{L}(C) \models_{\mathsf{P}} \varphi \wedge \mathcal{L}(D) \models_{\mathsf{P}} \varphi\big) \Rightarrow \big(\mathcal{L}(C \cap D) \models_{\mathsf{P}} \varphi \wedge \mathcal{L}(C \cup D) \models_{\mathsf{P}} \varphi\big)$$

Regular predicates were extensively studied by Garg, Mittal and Sen in the context of *computation slicing*, see e.g. [Garg and Mittal, 2001a; Sen and Garg, 2003; Garg and Mittal, 2005]. The main idea behind computation slicing is to try and reduce the size of a distributed execution, and therefore the number of cuts to be examined, in order to detect a predicate. For that, the authors use the duality between partial orders and distributive lattices characterized by Birkhoff's representation theorem, i.e. Theorem 1.2. Indeed, they first observe that the lattice of cuts is distributive.

Theorem 5.1 (The Lattice of Cuts is Distributive [Garg and Mittal, 2005])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, the lattice of cuts $\langle Q, \subseteq \rangle$ is distributive.

Therefore, given a predicate φ, they try to compute a poset that characterises, using Birkhoff's duality theorem, the smallest sublattice of the lattice of cuts that contains $[\![\varphi]\!]^T_{\mathsf{P}}$. In practice, instead of posets, the authors use directed graphs to represent sublattices of the lattice of cuts. We introduce the notion of *computation graph* to capture this in our framework.

Definition 5.8 (Computation Graph)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes, a computation graph of T is a directed graph $\langle N, R \rangle$ such that:

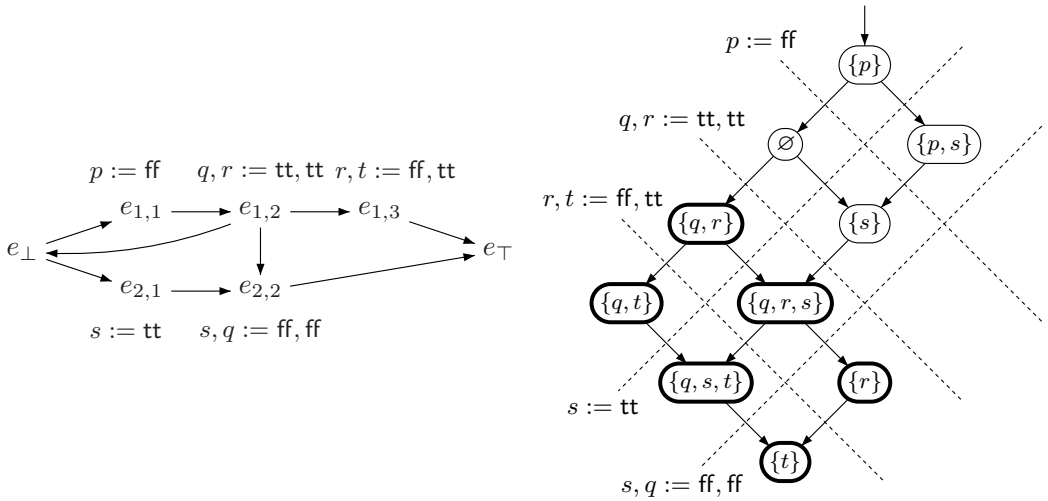
(i) $N = E \cup \{e_\top, e_\bot\}$

(ii) $\forall i \in [1, k] : \{\langle e_\bot, \mathsf{min}_\preceq(P_i)\rangle, \langle \mathsf{max}_\preceq(P_i), e_\top\rangle\} \subseteq R$

(iii) $\preceq \subseteq (R)^*$

In other words, a computation graph is a directed graph such that (i) there is one node for every event of T and two additional nodes $e_\top$ and $e_\bot$ representing respectively the beginning and the end of the trace[2], (ii) for all processes P_i, there is an edge from $e_\bot$ to the first event of P_i, and an edge from the last event of P_i to $e_\top$ and finally (iii) the set of edges contains at least the edges of the Hasse Diagram of the trace.

[2]Those two nodes are introduced to simplify the slicing algorithm



(a) a computation graph of the po-trace of Figure 4.4

(b) the corresponding lattice (in bold) of consistent cuts

Figure 5.4 - Example of computation graph

Given a po-trace T, we note $\mathsf{Graphs}(T)$ the set of computation graphs of T. Note that contrarily to Hasse diagrams, a computation graph may contain cycles. Intuitively, similarily to posets, a computation graph $\langle N, R\rangle$ represents a subset of cuts of the po-trace, i.e. the cuts closed w.r.t. R. Those cuts are called *consistent* cuts. Formally, given a computation graph $\langle N, R\rangle$, a set of nodes $M \subseteq N$ is *closed* w.r.t. R, if and only if $\forall e, e' \in N : ((e\,R\,e') \wedge (e' \in M)) \Rightarrow (e \in M)$. A cut C of T is *consistent* with a $\langle N, R\rangle$ if and only if $C \cup \{e_\perp\}$ is closed w.r.t. R. In the following, the set of cuts consistent with a computation graph $\langle N, R\rangle \in \mathsf{Graphs}(T)$ is noted $\mathsf{CC}(N, R)$. Another way to look at this, is to view each strongly connected component as a *meta-event*, i.e. a set of events that have been "glued" together, and should therefore be triggered in one step atomically. Indeed, by definition, a consistent cut C either contains all the events in such strongly connected component, or none of them.

Example 5.11

Figure 5.4(a) shows a computation graph $\langle N, R\rangle$ of the po-trace of Figure 4.4 (page 95). The corresponding lattice (in bold) of consistent cuts is presented in Figure 5.4(b). Note e.g. that the initial empty cut, i.e. the one labelled with $\{p\}$, is not consistent with. $\langle N, R\rangle$. Indeed $\varnothing \cup \{e_\perp\} = \{e_\perp\}$ is not closed w.r.t. R, since $e_{1,2}\,R\,e_\perp$ and $e_{1,2} \notin \{e_\perp\}$.

As shown in [Garg and Mittal, 2005], the set of cuts consistent with a computation graph along with set inclusion, also forms a distributive lattice.

Theorem 5.2 (Lattice of Consistent Cuts [Garg and Mittal, 2005])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and a computation graph $\langle N, R \rangle \in \mathsf{Graphs}(T)$, we have that $\langle \mathsf{CC}(N, R), \subseteq \rangle$ forms a complete distributive lattice.

The main idea behind computation slicing is therefore to build a computation graph representing exactly the sublattice of cuts that satisfy a regular predicate φ. Such a computation graph is called a *slice*, and is formalized as follows.

Definition 5.9 (Slice [Garg and Mittal, 2005])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and a predicate φ, a slice of T w.r.t. φ is a computation graph $\langle N, R_\varphi \rangle \in \mathsf{Graphs}(T)$ such that:

(i) $[\![\varphi]\!]^T_{\mathsf{P}} \subseteq \mathsf{CC}(N, R_\varphi)$

(ii) $\forall \langle N, R \rangle \in \mathsf{Graphs}(T) : ([\![\varphi]\!]^T_{\mathsf{P}} \subseteq \mathsf{CC}(N, R)) \Rightarrow (|\mathsf{CC}(N, R_\varphi)| \leq |\mathsf{CC}(N, R)|)$

Intuitively, a slice of a trace T w.r.t a predicate φ is a computation graph $\langle N, R_\varphi \rangle$ such that (i) every cut of T that satisfies φ is consistent with $\langle N, R_\varphi \rangle$ and (ii) among all those graphs, $\langle N, R_\varphi \rangle$ is one with the smallest number of consistent cuts. Note that this graph is not unique. However, two slices of a po-trace T w.r.t the same predicate φ have the same set of consistent cuts.

Theorem 5.3 (Uniqueness of Consistent Cuts in Slices [Garg and Mittal, 2005])

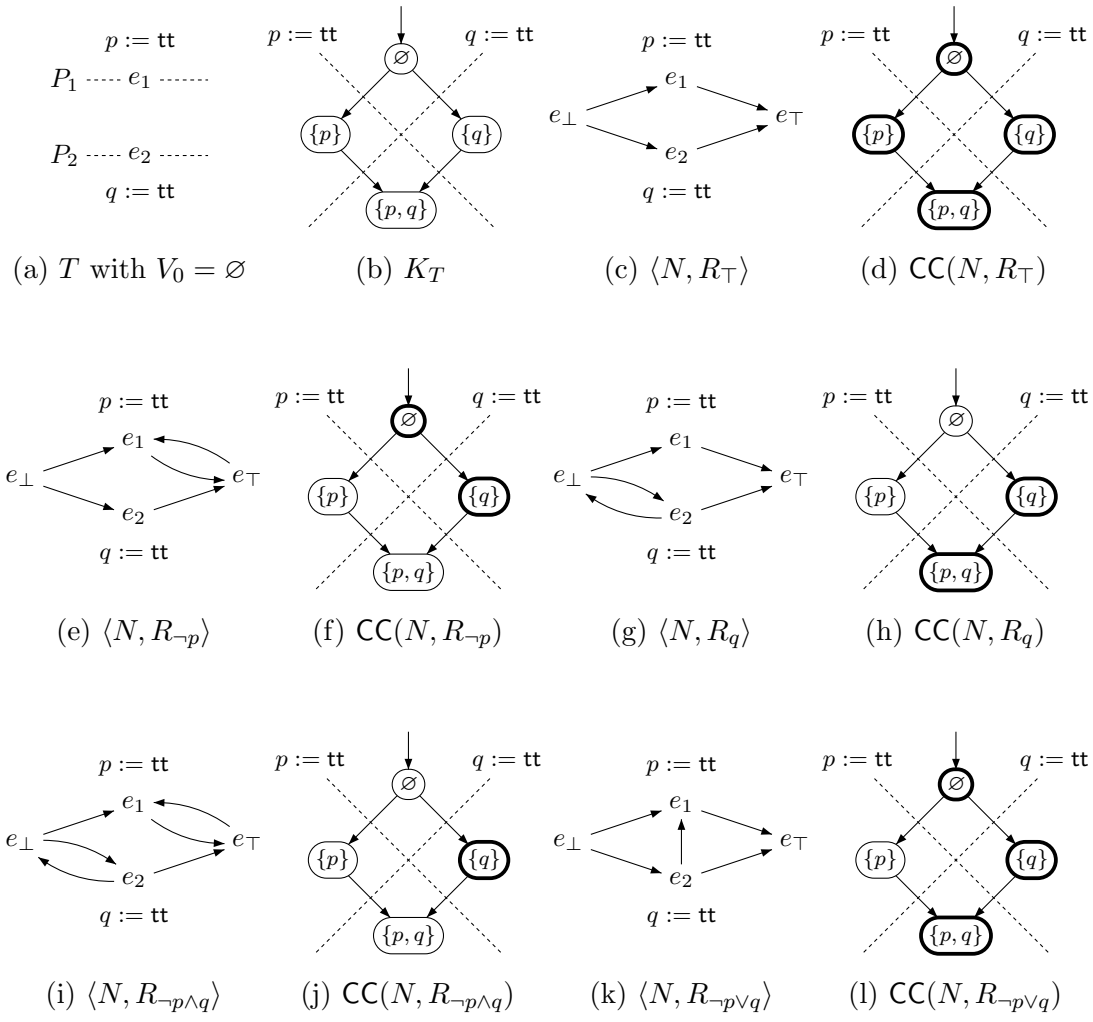
Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$, a predicate φ and two slices $\langle N, R_\varphi \rangle, \langle N, R'_\varphi \rangle$ of T w.r.t. φ, we have that $\mathsf{CC}(N, R_\varphi) = \mathsf{CC}(N, R'_\varphi)$.

A slice of T w.r.t φ can be computed inductively on the structure of φ, by composing slices of the subformulae of φ, as explained in [Garg and Mittal, 2005]. This is presented in Algorithm 5.8. For simplicity of the presentation, we assume that φ is in negation normal form. For $\top$ (lines 3–4), the slice is built by taking the Hasse Diagram (see Section 1.2) of $\preceq$, including the two special nodes $e_\top$ and $e_\bot$. For $\bot$ (lines 5–6), the slice is computed by taking the slice of $\top$ and adding an edge from $e_\top$ to $e_\bot$. Indeed, this forces any cut to be inconsistent with the slice. For literals (lines 7–13), the slice is built as follows. We starts with the slice of T w.r.t. $\top$. Then edges are added from every node that sets φ to false to the previous one that sets it to true. Special attention needs to be taken at the beginning and the end of the trace. This is taken care of by considering that $e_\top$ sets φ to true, and that $e_\bot$ set φ to false, if φ is false in V_0. Next, for conjunction (lines 14–15), the slice is obtained by taking the union of the set of

```
1  function computeSlice(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a predicate φ in NNF
   returns: R_φ ⊆ N × N such that ⟨N, R_φ⟩ is a slice of T w.r.t φ
2  begin
3      if φ = ⊤ then
4          R_φ := {⟨e, e'⟩ ∈ N × N | (e ≺ e') ∧ (∄e'' ∈ E : (e ≺ e'') ∧ (e'' ≺ e'))}
5      else if φ = ⊥ then
6          R_φ := computeSlice(T, ⊤) ∪ {⟨e_⊤, e_⊥⟩}
7      else if (φ = p) ∨ (φ = ¬p) then
8          R_φ := computeSlice(T, ⊤), b := (φ = p)
9          N_φ := {e ∈ E | p ∈ δ(e, b)} ∪ {e_⊤}
10         N_¬φ := {e ∈ E | p ∈ δ(e, b̄)}
11         if V_0 ⊭ φ then N_¬φ := N_¬φ ∪ {e_⊥}
12         forall e ∈ N_¬φ do
13             R_φ = R_φ ∪ {⟨min_⪯(N_φ \ ↓e), e⟩}
14     else if φ = φ_1 ∧ φ_2 then
15         R_φ := computeSlice(T, φ_1) ∪ computeSlice(T, φ_2)
16     else if φ = φ_1 ∨ φ_2 then
17         R_φ1 := computeSlice(T, φ_1), R_φ2 := computeSlice(T, φ_2)
18         R_φ := R⁺_φ1 ∩ R⁺_φ2
19     return R_φ
20 end
```

Algorithm 5.8 - Computation of a slice of a po-trace w.r.t a predicate

edges in the slices of both branches. Finally, for disjunction (lines 16–18), the slice is obtained by first computing the set of edges for the slices of the two branches, and then take the intersection of their transitive closure. Assuming that the Hasse Diagram of $\preceq$ is directly available, the slices in the first two cases can be obtained in $O(1)$. For literals, each event is considered at most once. The slice can therefore be built in $O(|E|)$. For the two remaining cases, the authors of [Garg and Mittal, 2005] provide an efficient way to compute R_φ by using a clever representation for slices, and show that using this method, the slice is computed in $O(|E| \times k^2)$. Since one slice is built for each subformula of φ, we can therefore deduce that the overall time complexity of Algorithm 5.8 is in $O(|E| \times |\varphi| \times k^2)$



(a) T with $V_0 = \varnothing$ (b) K_T (c) $\langle N, R_\top \rangle$ (d) $\mathsf{CC}(N, R_\top)$

(e) $\langle N, R_{\neg p} \rangle$ (f) $\mathsf{CC}(N, R_{\neg p})$ (g) $\langle N, R_q \rangle$ (h) $\mathsf{CC}(N, R_q)$

(i) $\langle N, R_{\neg p \wedge q} \rangle$ (j) $\mathsf{CC}(N, R_{\neg p \wedge q})$ (k) $\langle N, R_{\neg p \vee q} \rangle$ (l) $\mathsf{CC}(N, R_{\neg p \vee q})$

Figure 5.5 - Algorithm 5.8 at work

Example 5.12

Figure 5.5 illustrates the computation slicing algorithm presented above on a simple po-trace T composed of two events. First, the trace and its semantics are presented respectively in Figure 5.5 (a) and Figure 5.5 (b). Next, Figure 5.5 (c) and Figure 5.5 (d) respectively show the slice of T w.r.t. $\top$ obtained using Algorithm 5.8, and the corresponding consistent cuts. Figure 5.5 (e) to Figure 5.5 (h) show the slices of T w.r.t. $\neg p$ and q obtained using Algorithm 5.8, along with their respective sets of consistent cuts. Figure 5.5 (i) and Figure 5.5 (j) show the slices of T w.r.t. $\neg p \wedge q$ obtained using Algorithm 5.8 and the corresponding consistent cuts. Note that this slice was obtained by taking the union of the edges of the slices of Figure 5.5 (e) and Figure 5.5 (g). Finally, Figure 5.5 (i) and Figure 5.5 (j) show the slices of T w.r.t. $\neg p \vee q$ obtained using Algorithm 5.8 and the corresponding consistent cuts. In this slice, note that the edge $\langle e_2, e_1 \rangle$ is contained in both $R^+_{\neg p}$ and R^+_q.

Now that we have a way to compute the slice of T w.r.t. to a predicate, we can address the main issue, i.e. the detection of regular predicates. Indeed, it was proven in [Garg and Mittal, 2005] that the slice of a po-trace w.r.t. a predicate φ is regular if and only if any slice of T contains exactly those cuts that satisfy φ.

Theorem 5.4 (Slicing w.r.t Regular Predicates [Garg and Mittal, 2005])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and a predicate φ and a slice $\langle N, R_\varphi \rangle$ of T w.r.t. φ, we have that φ is regular w.r.t. T if and only if:

$$\mathsf{CC}(N, R_\varphi) = [\![\varphi]\!]^T_{\mathsf{P}}$$

Based on this observation, the authors of [Garg and Mittal, 2005] introduce a very elegant algorithm for the detection of regular predicates, formalized in Algorithm 5.9. All that needs to be done, is to compute the slice of the po-trace w.r.t. the regular predicate φ (line 3). Then, using Theorem 5.4, we know that there exists a cut satisfying φ if and only if this slice contains at least one consistent cut. In practice, this can be checked efficiently by considering the nodes $e_\top$ and $e_\bot$ (line 4). Indeed, it can be proven that the set of consistent cuts is empty if and only if there exists a path from $e_\top$ to $e_\bot$ in the slice. This can be done in $O(|E|)$. Hence the overall time complexity of Algorithm 5.9 is in $O(|E| \times |\varphi| \times k^2)$.

```
1 function detectRegular(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a regular predicate φ
  returns: tt if and only if T ⊨_P φ
2 begin
3 |   R_φ := computeSlice(T, NNF(φ))
4 |   return (⟨e_⊤, e_⊥⟩ ∈ R_φ^+)
5 end
```

Algorithm 5.9 - Detection of regular predicates

The authors of [Garg and Mittal, 2005] also show how a slice of a po-trace T w.r.t. an arbitrary predicate φ can be used for the detection of an arbitrary predicates φ. Indeed, we know that every cut satisfying φ is consistent with any slice $\langle N, R_\varphi \rangle$ of T w.r.t φ. Therefore, in order to determine if there exists a cut satisfying φ, it is only necessary to consider the set of cuts consistent with $\langle N, R_\varphi \rangle$. Algorithm 5.1 can be adapted for that purpose, using the strongly connected components of $\langle N, R_\varphi \rangle$. Indeed, if an event e belongs to a cut C, every other event in the connected component of e must belong to C in order for C to be consistent with $\langle N, R_\varphi \rangle$. Therefore, during the exploration, when triggering an event e from a cut C, instead of simply adding e to C, we add all events in the strongly connected component of e. The resulting algorithm is presented in Algorithm 5.10. In this algorithm, we assume the existence of a function $\mathsf{connectedComponent}(\langle N, R \rangle, n)$ that returns the strongly connected component of a directed graph $\langle N, R \rangle$ that contains the node $n \in N$. This algorithm starts by computing the slice of T w.r.t. φ (line 3). Then, it computes the strongly connected component of $e_\bot$ (line 4). If $e_\top$ belongs to this strongly connected component, then the set of consistent cut is empty and the algorithm returns false (lines 5–6). Indeed, in such a case, there is a path from $e_\top$ to $e_\bot$. Therefore, for any cut $C \in Q$, $C \cup \{e_\bot\}$ is not closed to R because $e_\top \notin R$. Otherwise, i.e. if $e_\top$ does not belong to the strongly connected component of $e_\bot$, the algorithm starts the exploration with a cut built from this strongly connected component. At each step, a consistent cut is considered. First, the algorithm checks whether the predicate holds in this cut (line 11–12). If this is the case, the algorithm returns true. Otherwise, the events enabled in the current cut are examined. However, in this case, when triggering an event e, instead of simply adding e as in the original algorithm, all the events in the strongly connected component of e are added (lines 14–16). If the loop terminates without finding a satisfying cut, the algorithm returns false (line 17). Decomposing $\langle N, R_\varphi \rangle$ into its strongly

```
1  function detectArbitrarySlicing(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, an arbitrary predicate φ
   returns: tt if and only if T ⊨_P φ
2  begin
3      N = E ∪ {e_⊤, e_⊥}, R_φ := computeSlice(T, NFF(φ))
4      M := connectedComponent(⟨N, R_φ⟩, e_⊥)
5      if e_⊤ ∈ M then
6          return ff;
7      W := {M \ {e_⊥}}, Z := ∅
8      while W ≠ ∅ do
9          Choose C ∈ W
10         W := W \ {C}, Z := Z ∪ {C}
11         if models(T, C, φ) then
12             return tt
13         forall e ∈ enabled(C) do
14             M := connectedComponent(⟨N, R_φ⟩, e)
15             if e_⊤ ∉ M then
16                 W := (W ∪ {(C ∪ M) \ {e_⊥}}) \ Z
17     return ff;
18 end
```

Algorithm 5.10 - Detection of arbitrary predicates using slicing

connected components can be done in $O(|N| + |R|)$ using Tarjan's algorithm [Tarjan, 1972]. Computing the slice is done in $O(|E| \times |\varphi| \times k^2)$. The exploration (lines 3–17) is done in $O(2^{|E|} \times |\varphi| \times k)$. Indeed, in a worst case scenario, each cut of T is consistent with the slice. Hence, the time complexity of Algorithm 5.10 is in $O(2^{|E|} \times |\varphi| \times k^2)$. However, in practice, as shown in [Garg and Mittal, 2005], it is substantially better than Algorithm 5.1.

Before concluding this section on regular predicates, let us investigate the relation of this class of predicates with the previous two. First, as proven in [Chase and Garg, 1998], the class of regular predicates is included in the class of linear predicates. To establish this result, the authors introduce the class of *meet-closed* predicate, i.e. predicates that are closed for the meet operator (intersection).

Definition 5.10 (Meet-Closed Predicate)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is meet-closed w.r.t. T if and only if

$$\forall C, D \in Q : \big(\mathcal{L}(C) \models_{\mathsf{P}} \varphi \wedge \mathcal{L}(D) \models_{\mathsf{P}} \varphi\big) \Rightarrow \mathcal{L}(C \cap D) \models_{\mathsf{P}} \varphi$$

From their respective definition, the authors observe that meet-closed predicates include regular predicates. Finally, they use the fact that a predicate is linear if and only if it is meet-closed.

Theorem 5.5 (Linear and Meet-Closed Coïncide [Chase and Garg, 1998])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a predicate φ is linear w.r.t. T if and only if it is meet-closed w.r.t. T

It follows directly that every regular predicate is also linear.

Corollary 5.2 ([Chase and Garg, 1998])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, any predicate φ regular w.r.t. T is also linear w.r.t T.

Furthermore, it is shown in [Garg and Mittal, 2001b] that regular predicates also include conjunctive predicates. To establish this result, the authors state, without proof, that local predicates are regular. We give a proof hereafter.

Proposition 5.8

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, any predicate φ local to a process P_i is also regular w.r.t. T.

Proof

Assume the existence of two cuts $C, D \in Q$ such that $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$, and $\mathcal{L}(D) \models_{\mathsf{P}} \varphi$. Since $\langle P_i, \preceq \rangle$ is a totally ordered set, we have that either $C \cap P_i \subseteq D \cap P_i$ or $D \cap P_i \subseteq C \cap P_i$. We only examine the former case, the latter being symmetrical. In the former case, we can deduce that $(C \cap D) \cap P_i = C \cap P_i$. Therefore, since $\mathcal{L}(C) \models_{\mathsf{P}} \varphi$, it follows, by Definition 5.1, that $\mathcal{L}(C \cap D) \models \varphi$. Similarly, in this case, we have that $(C \cup D) \cap P_i = D \cap P_i$. Again, since $\mathcal{L}(D) \models_{\mathsf{P}} \varphi$, we have, by Definition 5.1, that $\mathcal{L}(C \cup D) \models \varphi$.

Then, they prove that regular predicates are closed for conjunction.

Proposition 5.9 ([Garg and Mittal, 2001b])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, two predicates φ and ψ regular w.r.t. T, we have that $\varphi \wedge \psi$ is regular w.r.t T.

It follows directly that regular predicates include conjunctive predicates.

Corollary 5.3 ([Garg and Mittal, 2001b])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$, a conjunctive predicate φ is regular w.r.t. T.

5.2 A Syntactical Perspective

All the classes of predicates reviewed in the previous section are defined semantically. It is therefore not easy to detect whether a given PBL formula φ belongs to one of those classes without examining the lattice of cuts. In this section, we therefore try to come up with syntactical criteria for the two most important classes, i.e. observer-independent and regular. For that purpose, we focus our attention on literals, since they are the basic building blocks from which every predicate is built. It turns out that literals are observer-independent.

Theorem 5.6 (Litterals are Observer-Independent)

Given a set of propositions $\mathbb{P}$, a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a proposition $p \in \mathbb{P}$, we have that both p and $\neg p$ are observer-independent w.r.t. T

Proof

Let φ be a literal, i.e. either $\varphi = p$ or $\varphi = \neg p$ for some $p \in \mathbb{P}$ and assume that $\exists \rho \in \mathsf{runs}(K_T), \exists i \in [0, |\rho|) : \mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi$. We consider two cases:

(*i*) If $\rho(i)_{/p} = \varnothing$, by Definition 4.4, it must be that $V_0 \models_{\mathsf{P}} \varphi$. In this case, since $\varnothing$ is the first cut of every run and since $\mathcal{L}(\varnothing) = V_0$, we conclude.

(*ii*) On the other hand, if $\rho(i)_{/p} \neq \varnothing$, then the truth value of φ depends on one event $e \stackrel{\text{def}}{=} \mathsf{max}_{\preceq}(\rho(i)_{/p})$, and only on e. More precisely, assuming $b \in \mathbb{B}$ is a boolean value such that $b = \mathsf{tt}$ iff $\varphi = p$, we have that $\mathcal{L}(\rho(i)) \models_{\mathsf{P}} \varphi$ iff $p \in \delta(e, b)$. Then, for any run $\rho' \in \mathsf{runs}(K_T)$, there must exist a position $j \in [1, |\rho'|)$ such that $e \notin \rho'(j-1)$ and $e \in \rho'(j)$. Indeed, by definition, each event must must be triggered at some point during ρ since ρ ends in E. Since $\rho(j) \in Q = \mathsf{DC}_{\preceq}(E)$, it must be that $e = \mathsf{max}_{\preceq}(\rho'(j))$. It follows by Definition 4.4 that $\mathcal{L}(\rho'(j)) \models_{\mathsf{P}} \varphi$. Hence, we can conclude that $\forall \rho' \in \mathsf{runs}(K_T), \exists j \in [0, |\rho'|] : \mathcal{L}(\rho'(j)) \models_{\mathsf{P}} \varphi$.

It follows by Proposition 5.6, that disjunctions of literals are also observer-independent.

Corollary 5.4

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a PBL formula φ of the form $\varphi_1 \vee \ldots \vee \varphi_n$ where, $\forall i \in [1, n]$, φ_i is a literal, is observer-independent.

It also turns out that literals are regular. However, in order to prove that, we need a few intermediate results. First, given two cuts C and D of T such that $C_{/p} \subseteq D_{/p}$, we prove that the truth value of p in $C \cap D$ depends only on C.

Lemma 5.1

Given a set of propositions $\mathbb{P}$, a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a proposition $p \in \mathbb{P}$, and two cuts $C, D \in Q$ such that $C_{/p} \subseteq D_{/p}$, we have that $(\mathcal{L}(C) \models_{\mathsf{P}} p) \Leftrightarrow (\mathcal{L}(C \cap D) \models_{\mathsf{P}} p)$

Proof

We consider two cases:

(*i*) If $C_{/p} = \varnothing$, by Definition 4.4, we have that $\mathcal{L}(C) \models_{\mathsf{P}} p$ iff $p \in V_0$. By construction, we also have that $(C \cap D)_{/p} = C_{/p} \cap D_{/p} = \varnothing$. Therefore, by Definition 4.4, we also have that $\mathcal{L}(C \cap D) \models_{\mathsf{P}} p$ iff $p \in V_0$. Hence we conclude.

(*ii*) If $C_{/p} \neq \varnothing$, by Definition 4.4, we have that $\mathcal{L}(C) \models_{\mathsf{P}} p$ iff $p \in \delta(\mathsf{max}_{\preceq}(C_{/p}), \mathsf{tt})$. By construction, we have that $(C \cap D)_{/p} = C_{/p} \cap D_{/p} = C_{/p} \neq \varnothing$. It follows by Definitions 4.4 and 4.4, that $\mathcal{L}(C \cap D) \models_{\mathsf{P}} p$ holds iff $p \in \delta(\mathsf{max}_{\preceq}((C \cap D)_{/p}), \mathsf{tt})$. Finally, since $(C \cap D)_{/p} = C_{/p}$, we conclude.

Next, we prove that, in a similar situation, the truth value of any p in $C \cup D$ depends only on D.

Lemma 5.2

Given a set of propositions $\mathbb{P}$, a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a proposition $p \in \mathbb{P}$, and two cuts $C, D \in Q$ such that $C_{/p} \subseteq D_{/p}$, we have that $(\mathcal{L}(D) \models_{\mathsf{P}} p) \Leftrightarrow (\mathcal{L}(C \cup D) \models_{\mathsf{P}} p)$

Proof

We consider two cases:

(*i*) If $D_{/p} = \varnothing$, by Definition 4.4, we have that $\mathcal{L}(C) \models_{\mathsf{P}} p$ iff $p \in V_0$. By construction, we also have that $(C \cup D)_{/p} = C_{/p} \cup D_{/p} = C_{/p} \cup \varnothing$, and since $C_{/p} \subseteq D_{/p} = \varnothing$, that $(C \cup D)_{/p} = \varnothing$. Therefore, by Definition 4.4, we also have that $\mathcal{L}(C \cup D) \models_{\mathsf{P}} p$ iff $p \in V_0$. Hence we conclude.

(*ii*) If $D_{/p} \neq \varnothing$, by Definition 4.4, we have that $\mathcal{L}(C) \models_{\mathsf{P}} p$ iff $p \in \delta(\mathsf{max}_{\preceq}(D_{/p}), \mathsf{tt})$. By construction, we have that $(C \cup D)_{/p} = C_{/p} \cup D_{/p} = D_{/p} \neq \varnothing$. It follows by Definitions 1.2 and 4.4, that $\mathcal{L}(C \cup D) \models_{\mathsf{P}} p$ holds iff $p \in \delta(\mathsf{max}_{\preceq}((C \cup D)_{/p}), \mathsf{tt})$. Finally, since $(C \cup D)_{/p} = D_{/p}$, we conclude.

Finally, we prove that for any two cuts C and D of T, we are always in a situation where Lemma 5.1 or Lemma 5.2 applies.

Lemma 5.3

Given a set of propositions $\mathbb{P}$, a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we have that:

$$\forall C, D \in Q, \forall p \in \mathbb{P} : (C_{/p} \subseteq D_{/p}) \vee (D_{/p} \subseteq C_{/p})$$

Proof

First, we assume without loss of generality that $C_{/p} \neq \varnothing$ and that $D_{/p} \neq \varnothing$. Indeed, if one of the two, or both are empty, the result is immediate. In this case, since by Definition 4.4 both $C_{/p}$ and $D_{/p}$ are totally ordered w.r.t. $\preceq$, let $e \stackrel{\text{def}}{=} \mathsf{max}_{\preceq}(C_{/p})$ and $e' \stackrel{\text{def}}{=} \mathsf{max}_{\preceq}(D_{/p})$. Since both e and e' belong to $E_{/p}$, by Definition 4.4, we have that either $e \preceq e'$ or $e' \preceq e$. In the former case, for any $e'' \in C_{/p}$, we have that $e'' \preceq \mathsf{max}_{\preceq}(C_{/p}) = e \preceq e'$. Therefore, since $C \in Q = \mathsf{DC}_{\preceq}(E)$ and since $e' \in C$, we have that $e'' \in C$. Furthermore, since $e'' \in C_{/p}$, it must be that $e'' \in E_{/p}$ and therefore we have that $e'' \in D_{/p}$. We can therefore conclude that $C_{/p} \subseteq D_{/p}$. In the latter case, using a symmetrical argument, we can conclude that $D_{/p} \subseteq C_{/p}$ which proves the lemma.

We are now able to prove that literals are regular.

Theorem 5.7 (Litterals are Regular)

Given a set of propositions $\mathbb{P}$, a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over $\mathbb{P}$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a proposition $p \in \mathbb{P}$, we have that both p and $\neg p$ are regular w.r.t. T.

Proof

Let $C, D \in Q$ be two cuts of T. From Lemma 5.3, we known that either $C_{/p} \subseteq D_{/p}$ or $D_{/p} \subseteq C_{/p}$. Let us assume without loss of generality that $C_{/p} \subseteq D_{/p}$ (the other case is symmetrical). We consider p and $\neg p$ separately

$\boxed{p}$ Assume that $\mathcal{L}(C) \models_{\mathsf{P}} p$ and $\mathcal{L}(D) \models_{\mathsf{P}} p$. In this case, since $C_{/p} \subseteq D_{/p}$, we have by Lemma 5.1, that $\mathcal{L}(C \cap D) \models_{\mathsf{P}} p$, and by Lemma 5.2 that $\mathcal{L}(C \cup D) \models_{\mathsf{P}} p$. Hence, we conclude.

$\boxed{\neg p}$ Assume that $\mathcal{L}(C) \models_{\mathsf{P}} \neg p$ and $\mathcal{L}(D) \models_{\mathsf{P}} \neg p$. By Definition 1.2, this holds iff $\mathcal{L}(C) \not\models_{\mathsf{P}} p$ and $\mathcal{L}(D) \not\models_{\mathsf{P}} p$. Since $C_{/p} \subseteq D_{/p}$, we have by Lemma 5.1, that $\mathcal{L}(C \cap D) \not\models_{\mathsf{P}} p$, and by Lemma 5.2 that $\mathcal{L}(C \cup D) \not\models_{\mathsf{P}} p$. By Definition 1.2, this holds iff $\mathcal{L}(C \cap D) \models_{\mathsf{P}} \neg p$ and $\mathcal{L}(C \cup D) \models_{\mathsf{P}} \neg p$. Hence we conclude.

```
1 function detectDNF(T, φ)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a DNF predicate φ = φ_1 ∨ φ_2 ∨ ... ∨ φ_n
  returns: tt if and only if T ⊨_P φ
2 begin
3     forall i ∈ [1, n] do
4         if detectRegular(T, φ_i) then
5             return tt
6     return ff
7 end
```

Algorithm 5.11 - Detection of DNF predicates

Then, since regular predicates are closed to conjunction by Proposition 5.9, we can also conclude that conjunctions of literals are regular.

Corollary 5.5

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a PBL formula φ of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$ where, $\forall i \in [1, n]$, φ_i is a literal, is regular.

An important consequence of this result is that for a PBL formula φ in disjunctive normal form (DNF), i.e. φ is a disjunction of conjunctions of literals, detecting if there exists a cut satisfying φ can be done in polynomial time. Indeed, this can be reduced to checking if one of the branches of φ, i.e. a conjunction of literals, is satisfied by at least one cut. Since conjunctions of literals are regular, this can be done in polynomial time using Algorithm 5.9. This is formalized in Algorithm 5.11. It immediately follows that the predicate detection problem is PTIME-easy if the formula is in disjunctive normal form. This complexity gap can be explained by the fact transforming a formula into a DNF equivalent may increase its size by an exponential factor.

Theorem 5.8 (Complexity of PRED for PBL Formulae in DNF)

The predicate detection problem is PTIME-easy for any PBL formula in DNF.

This is a small, but rather interesting result. Indeed, in the context of testing, the predicate will be used to test a large number of po-traces. Therefore, the potentially exponential time needed to transform the formula in DNF will, in general, be greatly compensated by the time gained during the successive trace analyses.

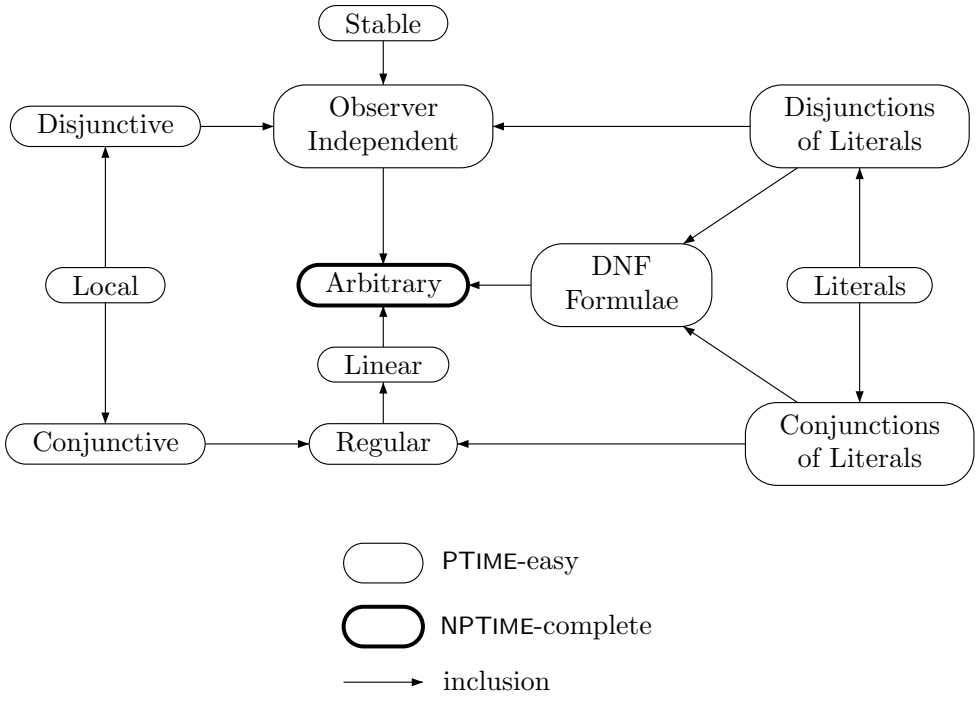


Figure 5.6 - A hierarchy of predicates

5.3 Summary

As a concluding remark, we present in Figure 5.6, an overview of all the classes of predicates that we examined in this chapter, along with their relationship with one another. At the center of this picture, stands the class of arbitrary predicates, defined by all PBL formulae, for which the detection problem is hard. Around those predicates, gravitate the well known classes of *stable*, *observer independent*, *local*, *disjunctive*, *conjunctive*, *regular* and *linear* predicates presented in Sections 5.1.1 through Section 5.1.7. Finally, on the right hand side of this picture, we can find the syntactical classes of predicates that we established in Section 5.2.

Chapter

6

LTL Trace Checking

« *Choice. The boy has no real choice, has he?* »

The prison chaplain, in *A Clockwork Orange*

In the linear case, the trace checking problem consists in determining if every linearisation of a distributed execution satisfies a given linear temporal requirement. In our framework, this translates into checking whether every trace σ of a given po-trace T satisfies, using the ad-hoc semantics, a given LTL formula φ. This problem is closely related to the LTL model checking problem. As previously mentioned, a classical approach to solve this problem is to use finite (Büchi) automata [Vardi and Wolper, 1986]. Indeed, as explained in Chapter 1, given a LTL formula φ, a Büchi automata $A_{\neg\varphi}$ accepting exactly $[\![\neg\varphi]\!]_{\mathsf{L}}$ is constructed. The model checking is then reduced to checking whether $\omega\text{-}\mathsf{traces}(K) \subseteq \omega\text{-}\mathsf{lang}(A_{\neg\varphi})$. In this chapter, we investigate how to adapt this for our purpose, i.e. solving the LTL trace checking problem, and examine if, in this particular context, it can be improved. First, in Section 6.1, we show how the classical automata-based approach, can be easily adapted. For that purpose, we show how the automaton construction can be modified to take into account the finite nature of partial order traces. We then explain how the resulting finite automaton, called a *monitor* in this context, can be used in practice to solve our problem of interest. Then, in Section 6.2, we introduce a new approach where the structure of the monitor is exploited to improve on the previous algorithm. Next, in Section 6.3, we compare the two approaches experimentally. Finally, we conclude, in Section 6.4, by discussing some related works. The content of this chapter is based on a joint work with Alexandre Genon and Thierry Massart, published in [Genon et al., 2006].

6.1 Automata-Based Approach

Let us examine how the automata-based approach can be adapted for the LTL trace checking problem. First, in Section 6.1.1, we explain how to build a finite automaton from a LTL formula φ, that accepts exactly those finite propositional sequences that satisfy φ. Then, in Section 6.1.2, we explain how this can be used, to solve the trace checking problem, by composing it with the po-trace. Finally, in Section 6.1.3, we explain how this composition can be refined into a trace checking algorithm.

6.1.1 Monitor

Our goal is to build an automaton accepting exactly the finite propositional sequences satisfying a given LTL formula φ, namely $[\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$. In this context, we call those finite automata *monitors*. The formal definition follows.

Definition 6.1 (Monitor)

Given a set of propositions $\mathbb{P}$, a monitor is a finite automaton $M = \langle S, s_0, F, \Sigma, \Delta \rangle$ with $\Sigma = 2^{\mathbb{P}}$.

In the following, we say that a monitor M is a monitor for φ if and only if $\mathsf{lang}(M) = [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$. The problem of building such a monitor for φ has already been studied in [Giannakopoulou and Havelund, 2001], where the authors adapt the classical algorithm of [Gerth et al., 1995] to finite propositional sequences. This algorithm works as follows. First, the LTL formula is transformed into *negation normal form* (NNF) by pushing the negations inward. For conjunctions and disjunctions, this can be done using De Morgan's laws[1]. For the X modalities, we can use the equivalence $(\neg \mathsf{X}\, \varphi) \Leftrightarrow ((\mathsf{X}\, \neg\varphi) \vee (\neg \mathsf{X}\, \top))$. Indeed, contrarily to the infinite case, where $(\neg \mathsf{X}\, \varphi) \Leftrightarrow (\mathsf{X}\, \neg\varphi)$ holds, in the finite case, we have to take into account the fact that $\neg \mathsf{X}\, \varphi$ can be evaluated at the end of a propositional sequence. In this case, $\neg \mathsf{X}\, \varphi$ holds, while $\mathsf{X}\, \neg\varphi$ does not. An easy way to compensate for that is to add $\neg \mathsf{X}\, \top$, which holds if and only if evaluated at the end of a trace. For the U modalities, we can use the dual modality V, introduced in [Gerth et al., 1995], such that $\neg(\varphi_1\, \mathsf{U}\, \varphi_2) \Leftrightarrow (\neg\varphi_1\, \mathsf{V}\, \neg\varphi_2)$. Similarly to PBL formulae, we note $\mathsf{NNF}(\varphi)$ the formula obtained from φ by applying those equivalences. Then, the algorithm constructs a graph where each node n is a 4-uple $\langle I, N, O, X \rangle$. The first component, I standing for *incoming*, is the set of nodes which have an incoming edge to n. The second component, N standing for *new*, is a set of LTL formulae that must hold in n, but have not been processed yet. The third component, O standing for

[1] $\neg(\varphi_1 \wedge \varphi_2) \Leftrightarrow (\neg\varphi_1 \vee \neg\varphi_2)$, $\neg(\varphi_1 \vee \varphi_2) \Leftrightarrow (\neg\varphi_1 \wedge \neg\varphi_2)$ and $\neg\neg\varphi \equiv \varphi$

old, is the set of LTL formulae that have already been processed, and finally, the last component, X standing for *next*, is a set of LTL formulae that must hold in all the immediate successors of n. At each step in the algorithm, a formula in the *new* field of the current node is processed, and moved to its *old* field. Once a node has been fully *expanded*, i.e. once all the formulae in its *new* field of a node have been processed, the node is stored in a set S for later use. This is presented in function $\mathsf{expand}(n, S)$ of Algorithm 6.1 which works as follows. First, it checks whether there are unprocessed formulae in the *new* field of the current node n (line 3). If it is not the case, the node n is fully processed, and ready to be added to the set of nodes S. If a similar node is already in S, i.e. a node with the same *old* and *next* fields (line 4), then the current node is merged with this node (line 5). If such a similar node does not exist, n is simply added to S (line 7). The new current node is then constructed from the *next* field of n. If the current node does contain some unprocessed formula φ, it is moved from its *new* field to its *old* field (line 9). Then, the node is processed. The processing depends on the structure of φ.

- If φ is a literal, $\top$ or $\bot$ (lines 10–12), the algorithm checks if the *old* field contains a contradiction[2], in which case the current node is discarded. If this is not the case, the expansion continues with the current node (line 12).
- If $\varphi = \mathsf{X}\,\varphi_1$ (lines 13–14), φ_1 is added to the *next* field.
- If $\varphi = \varphi_1 \wedge \varphi_2$ (lines 15–16), both φ_1 and φ_2 are added to the *new* field.
- If $\varphi = \varphi_1 \vee \varphi_2$ (lines 17–20), the current node is split in two. The formula φ_1 is added to the *new* field of one of its copy, and φ_2 to the other.
- If $\varphi = \varphi_1\,\mathsf{U}\,\varphi_2$ (lines 21–24), the node is split in two: for the first copy, φ_1 is added to the *new* field and $\varphi_1\,\mathsf{U}\,\varphi_2$ to the *next* field. For the other copy, only φ_2 is added to the *new* field. This splitting can be explained by observing that $\varphi_1\,\mathsf{U}\,\varphi_2$ is equivalent to $\varphi_2 \vee (\varphi_1 \wedge \mathsf{X}(\varphi_1\mathsf{U}\,\varphi_2))$.
- If $\varphi = \varphi_1\,\mathsf{V}\,\varphi_2$ (lines 25–28), the node is also split in two: for the first copy, φ_2 is added to the *new* field and $\varphi_1\,\mathsf{V}\varphi_2$ to the *next* field. For the other copy, only φ_1 is added to the *new* field. This splitting can be explained by observing that $\varphi_1\,\mathsf{V}\,\varphi_2$ is equivalent to $\varphi_2 \wedge (\varphi_1 \vee \mathsf{X}(\varphi_1\mathsf{V}\,\varphi_2))$, which is also equivalent to $(\varphi_2 \wedge \varphi_1) \vee (\varphi_2 \wedge \mathsf{X}(\varphi_1\,\mathsf{V}\,\varphi_2))$.

As shown in function $\mathsf{buildMonitor}(\varphi)$, the expansion starts with one node (line 34), containing the initial formula $\mathsf{NNF}(\varphi)$ in its *new* field, nothing in its *old* and *next* fields

[2]In the algorithm, we identify φ and $\neg\neg\varphi$

```
1  function expand(n, S)
   input  : a node n = ⟨I, N, O, X⟩ and the current set of nodes S
   output : S updated after the expansion of n
2  begin
3      if N = ∅ then
4          if ∃n' = ⟨I', N', O', X'⟩ ∈ S : (O = O') ∧ (X = X') then
5              return S \ {n'} ∪ {⟨I' ∪ I, N', O', X'⟩}
6          else
7              return expand(⟨{n}, X, ∅, ∅⟩, S ∪ {n})
8      else
9          Let φ ∈ N, N := N \ {φ}, O := O ∪ {φ}
10         if (φ = ⊤) ∨ (φ = ⊥) ∨ (φ = p) ∨ (φ = ¬p) then
11             if (φ = ⊥) ∨ (¬φ ∈ O) then  return S
12             else  return expand(⟨I, N, O, X⟩, S)
13         else if φ = X φ1 then
14             return expand(I, N, O, X ∪ {φ1})
15         else if φ = φ1 ∧ φ2 then
16             return expand(⟨I, N ∪ {φ1, φ2} \ O, O, X⟩, S)
17         else if φ = φ1 ∨ φ2 then
18             n1 := ⟨I, N ∪ {φ1} \ O, O, X⟩
19             n2 := ⟨I, N ∪ {φ2} \ O, O, X⟩
20             return expand(n2, expand(n1, S))
21         else if φ = φ1 U φ2 then
22             n1 := ⟨I, N ∪ {φ1} \ O, O, X ∪ {φ}⟩
23             n2 := ⟨I, N ∪ {φ2} \ O, O, X⟩
24             return expand(n2, expand(n1, S))
25         else if φ = φ1 V φ2 then
26             n1 := ⟨I, N ∪ {φ2} \ O, O, X ∪ {φ}⟩
27             n2 := ⟨I, N ∪ {φ1, φ2} \ O, O, X⟩
28             return expand(n2, expand(n1, S))
29 end
```

Algorithm 6.1 - Construction of monitors for LTL formulae

```
30 function buildMonitor(φ)
   input  : a LTL formula φ
   output : a monitor M such that lang(M) = [[φ]]^F_L
31 begin
32     ψ = NNF(φ)
33     n_0 := ⟨∅, ∅, ∅, {ψ}⟩
34     S := expand(⟨{n_0}, {ψ}, ∅, ∅⟩, {n_0})
35     Δ := ∅
36     forall n' = ⟨I', N', O', X'⟩ ∈ S do
37         forall V ⊆ P do
38             if ∀χ ∈ O' ∩ {p, ¬p | p ∈ P} : V ⊨_P χ then
39                 Δ := Δ ∪ {⟨n, V, n'⟩ ∈ S × 2^P × S | n ∈ I'}
40     F := {⟨I, N, O, X⟩ ∈ S | X = ∅}
41     return reduce(⟨S, n_0, F, 2^P, Δ⟩)
42 end
```

Algorithm 6.1 - Construction of monitors for LTL formulae (cont'd)

and the initial node n_0 in its *incoming* field. Once the expansion is finished, the monitor is built from the resulting set of nodes. The set of states is given by set of expanded nodes, to which is added the initial state (line 34). The transition relation is built as follows: there is an edge from a node n' to a node n labelled with a valuation $V \subseteq \mathbb{P}$ if and only if n' is in the *incoming* field of n, and if V is compatible with the literals in the *old* field of n (lines 35–39). The set of final states is then given by the set of nodes for which the *next* field is empty (line 40). In other words, a state is final if all future requirements are already satisfied. The resulting automaton is then determinized and minimized using the classical automata theory algorithm. For that purpose, we assume the existence of a function reduce (line 41) that takes care of that. For more details on how to minimize and determinize finite automata, we refer the reader to [Hopcroft et al., 2000].

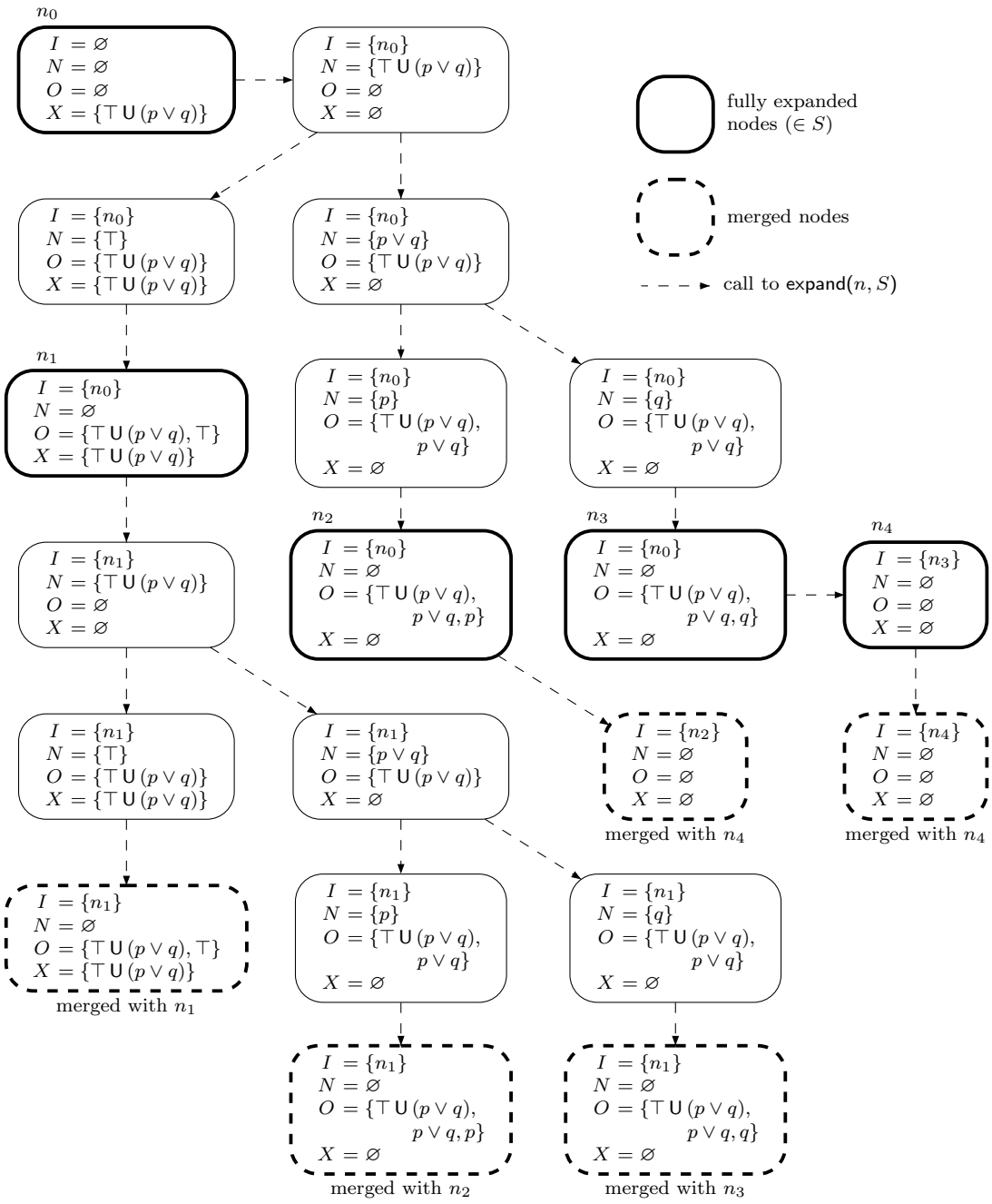
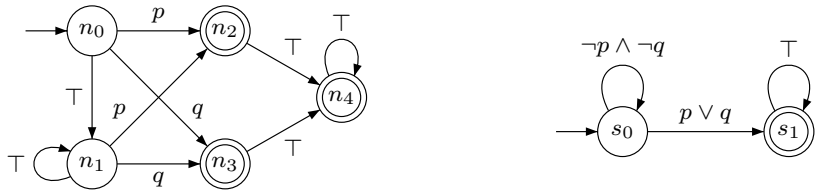


Figure 6.1 - Construction of a monitor for $\mathsf{F}(p \vee q)$ - graph expansion



(a) Monitor before reduction (b) Monitor after reduction

Figure 6.2 - Construction of a monitor for $\mathsf{F}\,(p \vee q)$ - reduction

Example 6.1

Figures 6.1 and 6.2 illustrate Algorithm 6.1 on the formula $\mathsf{F}(p \vee q)$. The expansion of the graph is presented in Figure 6.1. The resulting monitor, before and after reduction, is presented respectively in Figure 6.2(a) and Figure 6.2(b). For the sake of clarity, transitions between pair of states have been grouped together using PBL formulae, i.e. $n \xrightarrow{\varphi} n'$ denote $\{\langle n, V, n' \rangle \in Q \times 2^{\mathbb{P}} \times Q \mid V \models_{\mathsf{P}} \varphi\}$.

6.1.2 Composition

Similarly to what is done in automata-based model checking, using monitors, we can reduce the trace checking problem for LTL to inclusion checking. Formally, given a po-trace T and a monitor M_φ for a given LTL formula φ, we have to check that $\mathsf{traces}(K_T) \subseteq \mathsf{lang}(M_\varphi)$. Indeed, by doing so, we check that every trace of K_T satisfies φ. Again, similarly to automata-based model checking, we can alternatively check that $\mathsf{traces}(K_T) \cap \overline{\mathsf{lang}(M_\varphi)} = \varnothing$, or that equivalently that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M_{\neg\varphi}) = \varnothing$, where $M_{\neg\varphi}$ is a monitor for $\neg\varphi$. Testing emptiness bewteen the set of traces of a po-trace T and a monitor M can be done by composing T with M, i.e. examine how the monitor M reacts to the events of T. Formally, this composition is done as follows.

Definition 6.2 (Composition)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$, the composition of T with M, noted $T \times M$, is a transition system $\langle Q_\times, I_\times, \rightarrow_\times \rangle$ where:

- $Q_\times \stackrel{\text{def}}{=} Q \times S$,
- $I_\times \stackrel{\text{def}}{=} \{\langle \varnothing, s \rangle \mid s \in \mathsf{next}(s_0, V_0)\}$,
- $\rightarrow_\times \stackrel{\text{def}}{=} \{\langle \langle C, s \rangle, \langle C', s' \rangle \rangle \mid (C \rightarrow C') \wedge s' \in \mathsf{next}(s, \mathcal{L}(C'))\}$

In this composition, a configuration is a tuple $\langle C, s\rangle$, where C is a cut of T and s is a monitor state. The set of initial configurations is defined as the set of configurations $\langle \varnothing, s\rangle$, where s is a state that can be reached from the initial state s_0 of M by reading the initial valuation of T, namely V_0. This is because V_0 is the first valuation of all the traces of K_T, and should therefore be taken into account before any event is actually considered. Furthermore, the transition relation is defined as follows. In a configuration $\langle C, s\rangle$, when an event e is triggered, first its effect is taken into account in C, leading to a cut C', then the resulting valuation, i.e. $\mathcal{L}(C')$, is read by the monitor leading to a state s'. In essence, when in a configuration $\langle C, s\rangle$, we have that $\mathcal{L}(C)$ is the valuation that was used to reach s. With this definition, if $T \times M$ contains a run ending in a configuration $\langle C, s\rangle$ such that $C = E$ and $s \in F$, i.e. all events have been examined and the corresponding trace is accepted by M, then we know the $\mathsf{traces}(K_T)$ and $\mathsf{lang}(M)$ have a non-empty intersection. In fact, this is a necessary and sufficient condition.

Theorem 6.1 (Correctness of the composition)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$ and a monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta\rangle$, we have that:

$$(\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing) \Leftrightarrow (\forall \langle C, s\rangle \in \mathsf{reach}(T \times M) : (C = E) \Rightarrow (s \notin F))$$

Proof

We prove both directions:

$\boxed{\Rightarrow}$ Assume that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing$. Let $\langle C, s\rangle \in \mathsf{reach}(T \times M)$. If $C \neq E$, there is nothing to prove. Let us therefore consider the case where $C = E$. In this case, since $\langle C, s\rangle \in \mathsf{reach}(T \times M)$ and since $C = E$, there exists a run $\langle C_1, s_1\rangle\langle C_2, s_2\rangle \ldots \langle C_n, s_n\rangle \in \mathsf{runs}(T \times M)$ such that $\langle C_n, s_n\rangle = \langle C, s\rangle$. By Definition 6.2, we have that $C_1C_2 \ldots C_n \in \mathsf{runs}(K_T)$ and therefore that $\mathcal{L}(C_1C_2 \ldots C_n) \in \mathsf{traces}(K_T)$. Moreover, we also have that $\forall i \in [0, n) : \langle s_i, \mathcal{L}(C_{i+1}), s_{i+1}\rangle \in \Delta$, which implies that $\mathcal{L}(C_1C_2 \ldots C_n) \in \mathsf{lang}(M)$ if $s_n \in F$. However, since $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing$, it must be that $\mathcal{L}(C_1C_2 \ldots C_n) \notin \mathsf{lang}(M)$. It follows directly that $s = s_n \notin F$.

$\boxed{\Leftarrow}$ We proceed by contraposition. Assume indeed that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) \neq \varnothing$. In this case, there exists a run $\rho \in \mathsf{runs}(K_T)$ such that $\mathcal{L}(\rho) \in \mathsf{lang}(M)$. Therefore, there exists a sequence $s_0s_1 \ldots s_{|\rho|} \in S^+$ such that $\forall i \in [0, |\rho|) : \langle s_i, \mathcal{L}(\rho(i)), s_{i+1}\rangle$ with $s_{|\rho|} \in F$. By Definition 6.2, it follows directly that $\langle \rho(0), s_1\rangle\langle \rho(1), s_2\rangle \ldots \langle \rho(|\rho| - 1), s_{|\rho|}\rangle \in \mathsf{runs}(T \times M)$. We can therefore conclude that $\exists \langle C, s\rangle \in \mathsf{reach}(T \times M) : (C = E) \wedge (s \in F)$.

```
1  function explicitLTL-TC(T, φ)
   input  : A po-trace T = ⟨E, V_0, δ, ⪯⟩ and a LTL formula φ
   output : tt if and only if T ⊨_L φ
2  begin
3      ⟨S, s_0, F, 2^P, Δ⟩ := buildMonitor(¬φ)
4      Z := ∅, W := ∅
5      forall s ∈ next(s_0, V_0) do
6          W := W ∪ {⟨∅, V_0, s⟩}
7      while W ≠ ∅ do
8          Let ⟨C, V, s⟩ ∈ W
9          W := W \ {⟨C, V, s⟩}, Z := Z ∪ {⟨C, V, s⟩}
10         if (C = E) ∧ (s ∈ F) then
11             return ff
12         forall e ∈ enabled(C) do
13             V' := (V \ δ(e, ff)) ∪ δ(e, tt)
14             forall s' ∈ next(s, V')) do
15                 W := (W ∪ {⟨C ∪ {e}, V', s'⟩}) \ Z
16     return tt
17 end
```

Algorithm 6.2 - Explicit LTL trace checking algorithm

6.1.3 Explicit Trace Checking Algorithm

The necessary and sufficient condition of Theorem 6.1 can be refined into an explicit LTL trace checking algorithm, as presented in Algorithm 6.2. The algorithm starts by building the monitor for $\neg\varphi$ (line 3). The algorithm maintains two sets of configurations Z and W (line 4). The first set Z, initially empty, is used to remember the configurations that have already been explored in order to avoid exploring a configuration more that once. The other set W is the set of working configurations, i.e. the configurations that still need exploring. On top of a cut C and a monitor state s, each configuration also contains a valuation V corresponding to $\mathcal{L}(C)$. In the beginning, W is filled with the set of initial configurations as specified in Definition 6.2 (lines 5–6). Then, while the set of working configuration W is not empty, i.e. while there remains at least one configuration that has not been explored yet (line 7), the algorithm picks

one configuration arbitrarily in W (lines 8–9), and checks if it is a faulty configuration (line 10). If it is, the algorithm returns false (line 11) indicating that the formula is violated. On the other hand, if it is not, the algorithm adds to W every successor of the current configuration according to Definition 6.2 (lines 12–15). Finally, if the exploration stops without finding any faulty configurations, it returns true (line 16).

6.2 Monitor Driven Approach

In the classical automata-based approach presented in the previous section, every single interleaving of events compatible with the partial order of the po-trace is examined. Unfortunately, the number of such interleavings can be exponential in the number of events, which renders Algorithm 6.2 quite inefficient in practice. However, we can once again draw inspiration from the LTL model checking, and in particular from *partial order reduction* techniques [Godefroid, 1996; Valmari, 1993]. These techniques are based on the idea that a property is often insensitive to the order in which two *independent* actions occur. Therefore, only exploring one interleavings of those actions is enough. Regrettably, as is, those techniques are not very efficient for our problem of interest. Indeed, in the trace checking problem, the main model, namely partial order traces, is obtained from the system by collecting *only* events that are *relevant* to the property. In this setting, the property is therefore *dependent* to most, if not all, events of the po-trace, which is why partial order reduction brings almost no improvement. Nevertheless, we can still improve on the existing solution using the same underlying motivation, i.e. reducing the number of interleavings that need to be examined to establish the property. This is precisely what we address in this section. For that purpose, in Section 6.2.1, we first introduce a subclass of monitors, called *determined* monitors, which exhibit a remarkable property that will be useful for our purpose. We also show that every LTL formula admits a monitor in this subclass. Next, in Section 6.2.2, we explain how the structure of such a determined monitor can be exploited during the exploration. Finally, in Section 6.2.3, we show how this can be refined into a symbolic trace checking algorithm.

6.2.1 Determined Monitors

In the traditional composition, the effect of an event e on the monitor in a state s is determined not only by e, but also by the cut from which e is triggered. In this section, we introduce a subclass of monitors, called *determined* monitors, for which it is not the case, i.e. the effect of an event e on the monitor depends only on e. Intuitively, a

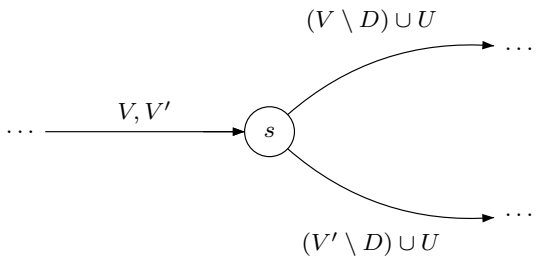


Figure 6.3 - $V \not\equiv_s V'$

monitor M is determined if and only if in any state s of M, all incoming valuations are equivalent for the evolution of the monitor. In the following, we say that two valuation $V, V' \subseteq \mathbb{P}$ are equivalent in a monitor state s, noted $V \equiv_s V'$, if and only if $\forall U, D \subseteq \mathbb{P} : \mathsf{next}(s, (V \setminus D) \cup U) = \mathsf{next}(s, (V' \setminus D) \cup U)$. In other words, as illustrated in Figure 6.3, V and V' are not equivalent in s, if there could exist an event, which effect is here modeled as a pair of sets of propositions U, D (U for *up* and D for *down*), that would result in different moves from s . Using this equivalence, we define the class of *determined* monitor as follows.

Definition 6.3 (Determined Monitor)

A monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$ is determined if and only if

$$\forall s \in S, \forall V, V' \in \mathsf{in}(s) : V \equiv_s V'$$

Before moving on, let us illustrate this definition with a small example.

Example 6.2

Consider the monitor presented in Figure 6.2(b). In the initial state s_0, the next move is determined only by the truth value of propositions p and q. However, for any incoming valuation $V \in \mathsf{in}(s_0)$, we have that $p \notin V$ and $q \notin V$. It follows that $\forall V \in \mathsf{in}(s_0) : V \equiv_{s_0} \varnothing$. In the final state s_1, the next move is always the same, i.e. coming back to s_1. Hence, all incoming valuations are equivalent in s_1. We can therefore conclude that this monitor is determined.

For this class of monitor, the set of monitor states reachable by triggering an event e from a monitor state s can be defined independently from the valuation that was used to reach s. Formally, we define $\mathsf{next}(s, e) \stackrel{\text{def}}{=} \{s' \in S \mid \exists V \in \mathsf{in}(s) : s' \in \mathsf{next}(s, (V \setminus \delta(e, \mathsf{ff})) \cup \delta(e, \mathsf{tt}))\}$, since all valuations are equivalent. Using this notation, we can give a simpler characterization of the transition relation when a po-trace is composed with a determined monitor.

Proposition 6.1

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$, let $T \times M = \langle Q_\times, I_\times, \rightarrow_\times \rangle$. We have that $\forall \langle C, s \rangle, \langle C', s' \rangle \in Q_\times$:

$$(\langle C, s \rangle \rightarrow_\times \langle C', s' \rangle) \Leftrightarrow (\exists e \in \mathsf{enabled}(C) : (C' = C \cup \{e\}) \wedge (s' \in \mathsf{next}(s, e)))$$

Of course, a natural question then arises: *Does every LTL formula admits such a determined monitor?* It turns out to be the case. In order to prove this, we exhibit an algorithm that, given an LTL formula φ, builds a determined monitor for φ from the one obtained using Algorithm 6.1. This algorithm is based on a necessary and sufficient condition for two valuations V, V' to be equivalent in a given monitor state s. For this, we need to introduce one additional notation. Given a monitor state s, we note $\mathsf{prop}(s) \stackrel{\text{def}}{=} \{p \in \mathbb{P} \mid \exists V \subseteq \mathbb{P} \setminus \{p\} : \mathsf{next}(s, V) \neq \mathsf{next}(s, V \cup \{p\})\}$ the set of propositions that directly affect the next move from s. At any given time during the composition, the only propositions that need to be taken into account when in a configuration $\langle C, s \rangle$ are the one in $\mathsf{prop}(s)$. This is formalized hereafter.

Proposition 6.2

Given a determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$, we have that:

$$\forall s \in S, \forall V \subseteq \mathbb{P} : \mathsf{next}(s, V) = \mathsf{next}(s, V \cap \mathsf{prop}(s))$$

Proof

We proceed by induction on $|V \setminus \mathsf{prop}(s)|$.

Initial Step If $|V \setminus \mathsf{prop}(s)| = 0$, then $V = V\mathsf{prop}(s)$ and the result is immediate.

Induction Step If $|V \setminus \mathsf{prop}(s)| = n > 0$, we have that $V \setminus \mathsf{prop}(s) \neq \varnothing$. Let $p \in V \setminus \mathsf{prop}(s)$, and $V' = V \setminus \{p\}$. By construction, since $p \notin \mathsf{prop}(s)$, we have that $V \cap \mathsf{prop}(s) = V' \cap \mathsf{prop}(s)$, and therefore that $\mathsf{next}(s, V \cap \mathsf{prop}(s)) = \mathsf{next}(s, V' \cap \mathsf{prop}(s))$. Then, since $|V' \setminus \mathsf{prop}(s)| = n - 1$, by induction, we have that $\mathsf{next}(s, V' \cap \mathsf{prop}(s)) = \mathsf{next}(s, V')$. Finally, since $p \notin \mathsf{prop}(s)$, and since $V' \subseteq (\mathbb{P} \setminus \{p\})$, by definition of $\mathsf{prop}(s)$, we have that $\mathsf{next}(s, V') = \mathsf{next}(s, V' \cup \{p\}) = \mathsf{next}(s, V)$. Hence, we conclude.

Example 6.3

Consider the monitor presented in Figure 6.2(b). In the initial state s_0, we have that $\mathsf{prop}(s) = \{p, q\}$, since those two propositions are the only one that are used to determine the next move. On the other hand, in the final state s_1, for any $V \subseteq \mathbb{P}$, we have that $\mathsf{next}(s_1, V) = \{s_1\}$. Hence, in this case, $\mathsf{prop}(s_1) = \varnothing$.

The necessary and sufficient condition is then stated as follows.

Theorem 6.2 (Necessary and Sufficient Conditions for Valuation Equivalence)

Given a monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$,we have that:

$$\forall s \in S : \forall V, V' \subseteq \mathbb{P} : \big(V \equiv_s V'\big) \Leftrightarrow \big(V \cap \mathsf{prop}(s) = V' \cap \mathsf{prop}(s)\big)$$

Proof

We prove both directions.

$\boxed{\Rightarrow}$ We proceed by contradiction. Assume indeed that $V \equiv_s V'$ and that $V \cap \mathsf{prop}(s) \neq V' \cap \mathsf{prop}(s)$. In this case, we have that $\exists p \in \mathsf{prop}(s)$ such that either $p \in V \setminus V'$ or $p \in V' \setminus V'$. We only consider the former case, the latter being symmetrical. Since $p \in \mathsf{prop}(s)$, we have that $\exists W \subseteq \mathbb{P} \setminus \{p\}$: $\mathsf{next}(s, W) \neq \mathsf{next}(s, W \cup \{p\})$. If we choose $U = W$ and $D = (V \cup V') \setminus \{p\}$, we have that $(V \setminus D) \cup U = \{p\} \cup W$ and that $(V' \setminus D) \cup U = \varnothing \cup W = W$. Hence, since $\mathsf{next}(s, W) \neq \mathsf{next}(s, W \cup \{p\})$. We therefore have that $V \not\equiv_s V'$ which contradicts our initial assumption.

$\boxed{\Leftarrow}$ Assume that $V \cap \mathsf{prop}(s) = V' \cap \mathsf{prop}(s)$. For any $U, D \subseteq \mathbb{P}$, we have that:

$$\begin{aligned}
\mathsf{next}(s, (V \setminus D) \cup U) &= \mathsf{next}(s, ((V \setminus D) \cup U) \cap \mathsf{prop}(s)) \\
&\qquad\qquad \text{(by Propositions 6.2)} \\
&= \mathsf{next}(s, ((V \cap \mathsf{prop}(M)) \setminus D) \cup (U \cap \mathsf{prop}(s))) \\
&= \mathsf{next}(s, ((V' \cap \mathsf{prop}(M)) \setminus D) \cup (U \cap \mathsf{prop}(s))) \\
&= \mathsf{next}(s, ((V' \setminus D) \cup U) \cap \mathsf{prop}(M)) \\
&= \mathsf{next}(s, (V' \setminus D) \cup U) \quad \text{(by Propositions 6.2)}
\end{aligned}$$

Hence, we can conclude that $V \equiv_s V'$.

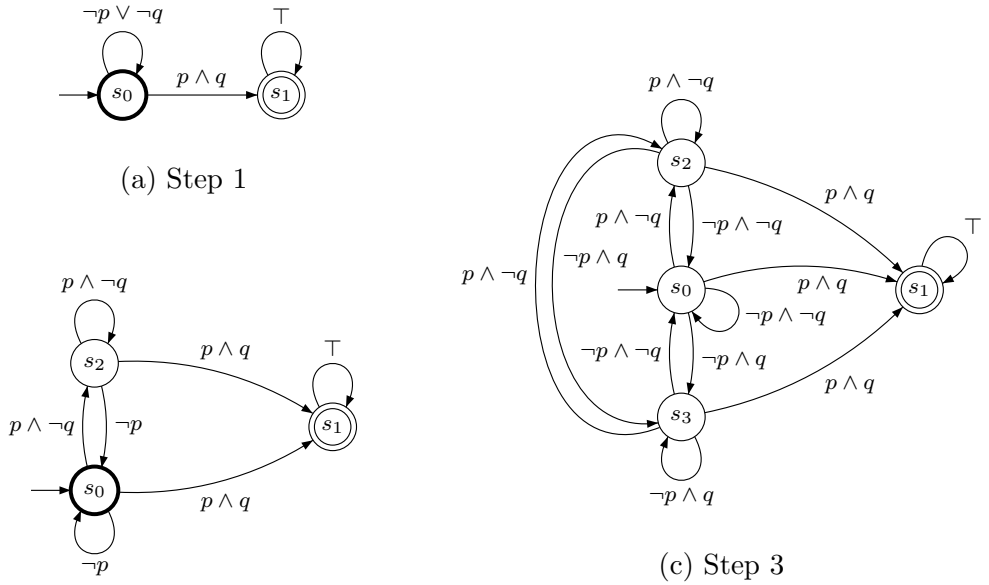
The algorithm is then formalized in Algorithm 6.3, which works as follows. It starts by building a monitor for φ using Algorithm 6.1 (line 3). Then, the algorithm loops as long as the monitor contains at least one state that does not comply with Definition 6.3, i.e. a state with two incoming valuations V, V' that are not equivalent (lines 4–13) according to the necessary and sufficient condition of Theorem 6.2. At each step of the loop, one of these faulty states is split. The copy s' takes on all the valuations in one of the equivalence classes induced by $\equiv_s$, while s will keep all the other ones. In practice, this splitting is done as follows. First, the state s' is created (line 5). If s is a final state, then s' is added to F (line 6). Then, all incoming valuations that are equivalent to V are diverted from s to s' (lines 8–11). Finally, in order to ensure that the language of the monitor is preserved, all outgoing transitions from s are also duplicated in s' (lines 12–13). The resulting monitor is then returned (line 14).

```
1  function buildDeterminedMonitor(φ)
   input  : a LTL formula φ
   returns: a determined monitor for φ
2  begin
3      ⟨S, s_0, F, 2^ℙ, Δ⟩ := buildMonitor(φ)
4      while ∃s ∈ S, ∃V, V' ∈ in(S) : (V ∩ prop(s) ≠ (V' ∩ prop(s)) do
5          Let s' be a new state, S := S ∪ {s'}
6          if s ∈ F then  F := F ∪ {s'}
7          X := {V'' ∈ in(s) | V'' ∩ prop(s) = V ∩ prop(s)}
8          forall V'' ∈ X s.t. ⟨s, V'', s⟩ ∈ Δ do
9              Δ := Δ ∪ {⟨s', V'', s'⟩}
10         forall r ∈ S, V'' ∈ X s.t. ⟨r, V'', s⟩ ∈ Δ do
11             Δ := (Δ \ {⟨r, V, s⟩}) ∪ {⟨r, V'', s'⟩}
12         forall t ∈ S, V'' ⊆ ℙ s.t. ⟨s, V'', t⟩ ∈ Δ do
13             Δ := Δ ∪ {⟨s', V'', t⟩}
14     return ⟨S, s_0, F, 2^ℙ, Δ⟩
15 end
```

Algorithm 6.3 - Construction of determined monitors for LTL formulae

Example 6.4

Figure 6.4 illustrates the construction of a determined monitor for the formula $\mathsf{F}(p \wedge q)$ using Algorithm 6.3. Faulty states are highlighted in bold. In this example, two splits are required to transform the original monitor into a determined one. First, Figure 6.4(a) shows the monitor obtained using Algorithm 6.1. In this automaton, the initial state s_0 is at fault because e.g. $\{p\} \not\equiv_{s_0} \varnothing$. Indeed, $\{p\} \cap \mathsf{prop}(s) = \{p\} \cap \{p, q\}, \{p\} \neq \varnothing \cap \mathsf{prop}(s) = \varnothing$. This state is therefore split. The copy s_2 takes the valuations equivalent to $\{p\}$, while the others remain in s_0. This leads to the automaton of Figure 6.4(b). In this automaton, s_0 is still at fault, since e.g. $\{q\} \not\equiv_{s_0} \varnothing$. Indeed, $\{q\} \cap \mathsf{prop}(s) = \{q\} \cap \{p, q\} = \{q\} \neq \varnothing \cap \mathsf{prop}(s) = \varnothing$. It is therefore split again. This time, the copy s_3 takes the valuations equivalent to $\{q\}$ and s_0 keeps the remaining valuations, i.e. the one equivalent to $\varnothing$. The resulting automaton, presented in Figure 6.4(c), meets the condition of Definition 6.3, and is therefore determined.



(a) Step 1

(b) Step 2

(c) Step 3

Figure 6.4 - Construction of a determined monitor for $\mathsf{F}(p \wedge q)$

We now prove the correctness of this algorithm.

Theorem 6.3 (Correctness of Algorithm 6.3)

Given an LTL formula φ, Algorithm 6.3 returns a determined monitor for φ

Proof

We successively prove partial correctness, i.e. that if the algorithm terminates, it returns the expected result, and then termination.

Partial Correctness An invariant for the main loop of the algorithm is that the automata is a monitor for φ. Indeed, at the beginning, the automata is obtained using Algorithm 6.1. Then, at each step of the loop, every propositional sequence σ accepted from s at the beginning of the body of the loop can be accepted by both s and its copy s' after the body. Then, since each transition to s is either redirected to s' or kept to s, the overall language of the automaton is preserved. This invariant, along with the looping condition implies that, if the loop terminates, the automaton is a determined monitor for φ.

Proof (cont'd)

Termination In a given state $s \in S$, let $P_s \in \Pi(\mathsf{in}(s))$ be the partition induced by $\equiv_s$. Let us then define the function $f \in S \mapsto \mathbb{N}$ as follows. If $|P_s| \leq 1$, then $f(s) \stackrel{\text{def}}{=} 0$, otherwise, $f(s) \stackrel{\text{def}}{=} |P_s|$. And let that $f(S) \stackrel{\text{def}}{=} \sum_{s \in S} f(s)$. By construction, we have that $f(S) \geq 0$. We prove that $f(S)$ strictly decreases after each execution of the body of the loop. Indeed, for any state $s'' \neq s$, before the body of the loop, the set of incoming valuations is not modified by the transformation. It follows that the contribution of s'' in $f(S)$ remains unchanged. For s, since one of the equivalence classes is redirected to s', $f(s)$ decreases by at least 1. Furthermore, since in s' all incoming valuations are equivalent $f(s') = 0$. We can therefore conclude that $f(S)$ strictly decreases at each step of the loop, thus ensuring termination.

It follows directly that every LTL formula admits a determined monitor

Theorem 6.4 (Existence of a Determined Monitor)

Given a LTL formula φ, there exists a determined monitor M such that

$$\mathsf{lang}(M) = [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$$

Moreover, from Theorem 6.3, we get a bound on the number of states that the resulting determined monitor may have. Indeed, in a worst case scenario, the number of splits required to transform M into a determined monitor is equal to $\sum_{s \in S} |\mathsf{in}(s)|$. Assuming d denotes the maximum input degree of M, i.e. $\mathsf{max}_{\leq}(\{|\mathsf{in}(s)| \mid s \in S\})$, we have that this number is bounded by $|S| \times d$. Finally, since d is bounded by $|\Delta|$, we have that the number of states of the resulting determined monitor is bounded by $|S| + |S| \times |\Delta| = |S| \times (|\Delta| + 1)$. Note however, that the transformation does not introduce additional non-determinism. Therefore, the size of the state space in the composition is not affected. More precisely if M denotes the original monitor and M' the resulting determined one, we have that for any po-trace T, there are as many reachable configurations in $T \times M$ as in $T \times M'$.

6.2.2 Monitor Driven Composition

We now explain how the structure of these determined monitors can be used during the exploration. This idea is based on the following observation. During the exploration, as illustrated in Figure 6.5, the monitor is not always *sensitive* to all events in its current state, i.e. some events have no effect on the monitor. Indeed, consider the event e

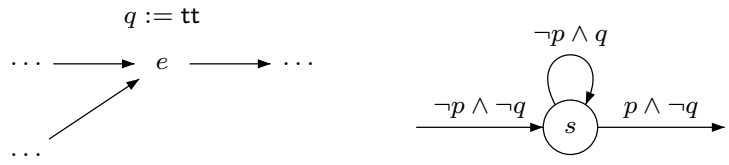


Figure 6.5 - Monitor sensitivity

and the monitor state s of Figure 6.5. In this state, we have that $\mathsf{next}(s, e) = \{s\}$, i.e. triggering e from any incoming valuation leads back to s. In other words, the monitor in not *sensitive* to e in s. Formally, given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$, and a determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$, the set of events of T to which M is sensitive in a state $s \in S$, noted $\mathsf{sensitive}(s)$, is defined as $\{e \in E \mid \mathsf{next}(s, e) \neq \{s\}\}$. In the classical exploration, when coming to a configuration $\langle C, s \rangle$, where a non-sensitive event could be, amongst others, triggered, two possibilities are considered: one where e is fired, and one where e is not. However, from the point of view of the monitor, both executions are identical. Hence, instead of considering those executions separately, we group them together into a *symbolic* execution, where e has *optionally* been triggered. During exploration, we therefore differentiate *necessary* events, i.e. events that triggered a monitor move, and the *optional* events, i.e. events that did not. Of course, after a monitor move, optional events might become sensitive and must therefore be re-examined, since the sensitivity of the monitor has changed. Concretely, this is captured in the *monitor driven* composition, defined hereafter.

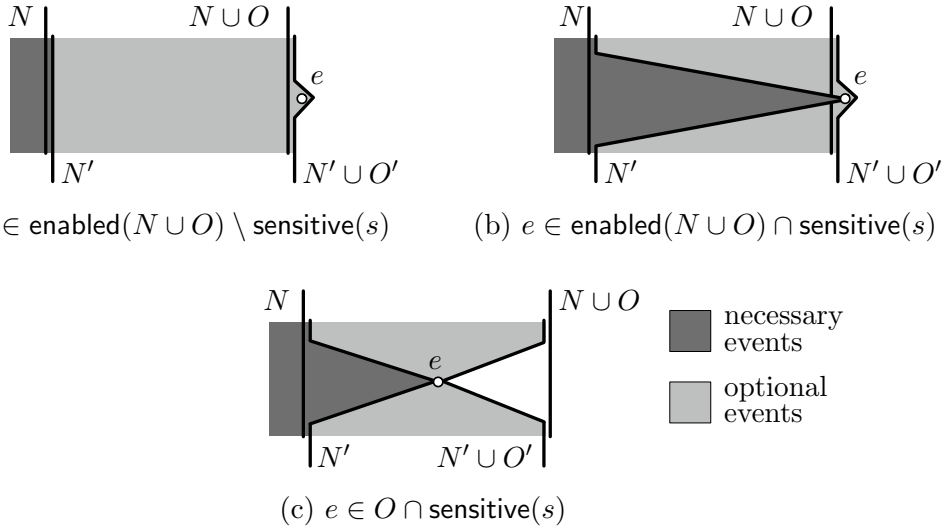
Definition 6.4 (Monitor Driven Composition)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta \rangle$, the monitor driven composition of T with M, noted $T \otimes M$, is a transition system $\langle Q_\otimes, I_\otimes, \rightarrow_\otimes \rangle$ where:

- $Q_\otimes \stackrel{\text{def}}{=} 2^E \times 2^E \times S$,
- $I_\otimes \stackrel{\text{def}}{=} \{\langle \varnothing, \varnothing, s \rangle \mid s \in \mathsf{next}(s_0, V_0)\}$,
- $\rightarrow_\otimes$ is s.t. $\langle N, O, s \rangle \rightarrow_\otimes \langle N', O', s' \rangle$ if and only if one of the following holds:

  (i) $\exists e \in \mathsf{enabled}(N \cup O) \setminus \mathsf{sensitive}(s)$:
  $$(N' = N) \wedge (O' = O \cup \{e\}) \wedge (s' = s)$$

  (ii) $\exists e \in (\mathsf{enabled}(N \cup O) \cup O) \cap \mathsf{sensitive}(s)$:
  $$(N' = N \cup {\downarrow}e) \wedge (O' = O \setminus ({\downarrow}e \cup {\uparrow}e)) \wedge (s' \in \mathsf{next}(s, e))$$



(a) $e \in \mathsf{enabled}(N \cup O) \setminus \mathsf{sensitive}(s)$ (b) $e \in \mathsf{enabled}(N \cup O) \cap \mathsf{sensitive}(s)$

(c) $e \in O \cap \mathsf{sensitive}(s)$

Figure 6.6 - Monitor driven composition

In Definition 6.4, each configuration is a tuple $\langle N, O, s\rangle$, where N is the set of *necessary* events, O is the set of *optional* events and s is the monitor state. As illustrated in Figure 6.6, the transitions relation is decomposed in two parts depending on the events e being triggered. The first case is when e a new event to which the monitor is not sensitive in its current state. In this case, as illustrated in Figure 6.6(a), e is simply added to the set of optional events. The second case is when e is a new or optional event to which the monitor is sensitive in its current state. In this case, as illustrated in Figure 6.6(c) and Figure 6.6(b), e, along with all its past is added to the set of necessary events. Moreover, e, along with all its past and future is removed from the set of optional events. The events in the past of e are removed because they are added to the set of mandatory events, whereas the events in its future are removed because they depended on the fact that e was triggered while the monitor was in the previous state. Each configuration in the monitor driven composition represents *symbolically* an entire set of configurations in the traditional composition. More precisely, each configuration $\langle N, O, s\rangle \in \mathsf{reach}(T \otimes M)$ represents $\{\langle C, s\rangle \in \mathsf{reach}(T \times M) \mid N \subseteq C \subseteq N \cup O\}$. This is proven in the following lemma.

Lemma 6.1

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$, and determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta\rangle$, we have that:

$$\forall \langle N', O', s'\rangle \in \mathsf{reach}(T \otimes M), \forall C \in Q : \left(\begin{array}{c} (N' \subseteq C' \subseteq N' \cup O') \Rightarrow \\ (\langle C', s'\rangle \in \mathsf{reach}(T \times M)) \end{array} \right)$$

Proof

We proceed by induction on the number n of transitions needed to reach $\langle N', O', s' \rangle$.

Initial case If $n = 0$, we have that $\langle N', O', s' \rangle \in I_{\otimes}$. By Definition 6.4, we therefore have that $N' = O' = \varnothing$ and that $s' \in \mathsf{next}(s_0, V_0)$. In this case, the only possibility is $C' = \varnothing$ and indeed, we have by Definition 6.2, that $\langle \varnothing, s' \rangle \in I_{\times} \subseteq \mathsf{reach}(T \times M)$.

Inductive case If $n > 0$, we have that $\langle N, O, s \rangle \rightarrow_{\otimes} \langle N', O', s' \rangle$ for some $\langle N, O, s \rangle \in \mathsf{reach}(T \otimes M)$ reachable in $n - 1$ tansitions. By induction, for any $C \in Q$ such that $N \subseteq C \subseteq N \cup O$, we have that $\langle C, s \rangle \in \mathsf{reach}(T \times M)$. Then, we examine the two possibilities of Definition 6.4:

(*i*) In the first case, $\exists e \in \mathsf{enabled}(N \cup O) \setminus \mathsf{sensitive}(s)$ such that $N' = N$, $O' = O \cup \{e\}$ and $s' = s$. Let $C' \in Q$ be a cut such that $N' \subseteq C' \subseteq N' \cup O'$. If $e \notin C'$, we have that $N \subseteq C' \subseteq N \cup O$ which implies by induction that $\langle C', s' \rangle \in \mathsf{reach}(T \times M)$. On the other hand, if $e \in C'$, by construction, we have that $e \in \mathsf{Max}_{\preceq}(C')$, which implies that $C' \setminus \{e\} \in Q$ and that $e \in \mathsf{enabled}(C' \setminus \{e\})$. Moreover, by construction, we have that $N \subseteq C' \setminus \{e\} \subseteq N \cup O$, which implies by induction that $\langle C' \setminus \{e\}, s \rangle \in \mathsf{reach}(T \times M)$. Furthermore, since $e \notin \mathsf{sensitive}(s)$, we have that $\mathsf{next}(s, e) = \{s\}$. It follows by Proposition 6.1 that $\langle C' \setminus \{e\}, s \rangle \rightarrow_{\times} \langle C', s \rangle = \langle C', s' \rangle$. Hence, we conclude.

(*ii*) In the second case, $\exists e \in (\mathsf{enabled}(N \cup O) \cup O) \cap \mathsf{sensitive}(s)$ such that $N' = N \cup \downarrow e$, $O' = O \setminus (\downarrow e \cup \uparrow e)$ and such that $s' \in \mathsf{next}(s, e)$. Let $C' \in Q$ be a cut such that $N' \subseteq C' \subseteq N' \cup O'$. By construction, since $C' \supseteq N' = N \cup \downarrow e$, we have that $e \in C'$. Moreover, since $C \subseteq N' \cup O' = (N \cup O) \setminus (\uparrow e \setminus \{e\})$, we have that $C' \cap \uparrow e = \{e\}$. It follows directly that $e \in \mathsf{Max}_{\preceq}(C')$, which implies that that $C' \setminus \{e\} \in Q$. We can therefore conclude that $e \in \mathsf{enabled}(C' \setminus \{e\})$. Moreover, by construction, $C' \supseteq N' \supseteq N$ and $e \notin N$, we have that $N = N \setminus \{e\} \subseteq C' \setminus \{e\}$. Also by construction, since $C' \subseteq O' \subseteq O$, we have that $C' \setminus \{e\} \subseteq O$. It follows, by induction, that $\langle C' \setminus \{e\}, s \rangle \in \mathsf{reach}(T \times M)$. Then, since $s' \in \mathsf{next}(s, e)$, by Proposition 6.1 we have that $\langle C' \setminus \{e\}, s \rangle \rightarrow_{\times} \langle C', s' \rangle$. Hence we conclude.

The previous lemma proves that the monitor driven composition is sound w.r.t. the classical composition. We also need to prove that it is complete, i.e. that every configuration $\langle C, s \rangle \in \mathsf{reach}(T \times M)$ is represented by at least one symbolic configuration in $\mathsf{reach}(T \otimes M)$. This is proven in the following lemma.

Lemma 6.2

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a determined monitor $M = \langle S, s_0, F, \Sigma, \Delta \rangle$, we have that:

$$\forall \langle C', s' \rangle \in \mathsf{reach}(T \times M), \exists \langle N', O', s' \rangle \in \mathsf{reach}(T \otimes M) : N' \subseteq C' \subseteq N' \cup O'$$

Proof

We proceed by induction on the number n of transitions needed to reach $\langle C', s' \rangle$.

Initial case If $n = 0$, we have that $\langle C', s' \rangle \in I_\times$. By Definition 6.2, we therefore have that $C' = \varnothing$ and that $s' \in \mathsf{next}(s_0, V_0)$. In this case, we can choose $N' = O' = \varnothing$, since, by Definition 6.4, $\langle \varnothing, \varnothing, s' \rangle \in \mathsf{reach}(T \otimes M)$.

Inductive case If $n > 0$, we have that $\langle C, s \rangle \rightarrow_\times \langle C', s' \rangle$ for some $\langle C, s \rangle \in \mathsf{reach}(T \times M)$ reachable in $n - 1$ transitions, with $C' \setminus C = \{e\}$. By induction, we have that $\exists \langle N, O, s \rangle \in \mathsf{reach}(T \otimes M)$ such that $N \subseteq C \subseteq N \cup O$. We have to examine the two possibilities:

(i) If $e \notin \mathsf{sensitive}(s)$, we have that $\mathsf{next}(s, e) = \{s\}$, which implies by Proposition 6.1 that $s' = s$. Then, either $e \in O$, in which case we have that $N \subseteq C' \subseteq N \cup O$ thus implying the result, or $e \notin O$. In this latter case, since $e \in \mathsf{enabled}(C)$, we have that $\downarrow e \setminus \{e\} \subseteq C \subseteq N \cup O$. It follows directly, since $e \notin C \supseteq N$, that $e \in \mathsf{enabled}(N \cup O)$. By Definition 6.4, we can therefore conclude that $\langle N, O, s \rangle \rightarrow \langle N, O \cup \{e\}, s \rangle$ with $N \subseteq C' \subseteq N \cup O \cup \{e\}$. Hence we conclude.

(ii) If $e \in \mathsf{sensitive}(s)$, we have that $s' \in \mathsf{next}(s, e)$ by Proposition 6.1. Then, using the same argument as in (i), we have that either $e \in O$ or $e \in \mathsf{enabled}(N \cup O)$. In both cases, we have that $e \in (\mathsf{enabled}(N \cup O) \cup O) \cap \mathsf{sensitive}(s)$, which implies by Definition 6.4 that $\langle N, O, s \rangle \rightarrow_\otimes \langle N', O', s' \rangle$ with $N' = N \cup \downarrow e$ and $O' = O \setminus (\downarrow e \cup \uparrow e)$. It is therefore left to prove that $N' \subseteq C'$ and that $C' \subseteq N' \cup O'$. For the former, we start with $N \subseteq C$, which implies that $N' = N \cup \downarrow e \subseteq C \cup \downarrow e$. Then, since $\downarrow e \subseteq C \cup \{e\}$, we have that $N' \subseteq C \cup \{e\} = C'$. For the latter, we start with $C \subseteq N \cup O$, which implies that $C' = C \cup \downarrow e \subseteq N \cup O \cup \downarrow e = (N \cup \downarrow e) \cup (O \setminus \downarrow e) = N' \cup (O \setminus \downarrow e)$. It follows that $C' \setminus (\uparrow e \setminus \{e\}) \subseteq (N' \cup (O \setminus \downarrow e)) \setminus (\uparrow e \setminus \{e\})$. Then we observe that $(\uparrow e \setminus \{e\}) \cap C' = \varnothing$, and since $N' \subseteq C'$ that $(\uparrow e \setminus \{e\}) \cap N' = \varnothing$. It follows directly that $C' \subseteq N' \cup (O \setminus (\downarrow e \cup (\uparrow e \setminus \{e\})))$. Finally, since $e \in \downarrow e$, we can conclude that $C' \subseteq N' \cup (O \setminus (\downarrow e \cup \uparrow e)) = N' \cup O'$.

Using the previous two lemmata, we are now able to establish the correctness of the monitor driven composition.

Theorem 6.5 (Correctness of the Monitor Driven Composition)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a determined monitor $M = \langle S, s_0, F, \Sigma, \Delta \rangle$, we have that:

$(\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing) \Leftrightarrow (\forall \langle N, O, s \rangle \in \mathsf{reach}(T \otimes M) : (N \cup O = E) \Rightarrow (s \notin F))$

Proof

We prove both directions:

- $\boxed{\Rightarrow}$ We proceed by contradiction. Assume that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing$ and that $\exists \langle N, O, s \rangle \in \mathsf{reach}(T \otimes M)$ with $N \cup O = E$ and $s \in F$. By Lemma 6.1, since $N \subseteq E = N \cup O$, this implies that $\langle E, s \rangle \in \mathsf{reach}(T \times M)$. It follows, by Theorem 6.1, that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) \neq \varnothing$ which contradicts our initial assumption.
- $\boxed{\Leftarrow}$ We proceed by contraposition. Assume indeed that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) \neq \varnothing$. By Theorem 6.1, we have that there exists a configuration $\langle C, s \rangle \in \mathsf{reach}(T \times M)$ such that $C = E$ and $s \in F$. It follows by Lemma 6.2 that there exists a configuration $\langle N, O, s \rangle \in \mathsf{reach}(T \otimes M)$ such that $N \subseteq C \subseteq N \cup O$. Since $C = E$ and $N \cup O \subseteq E$, it must be the case that $N \cup O = E$. Hence we conclude.

6.2.3 Symbolic Trace Checking Algorithm

If explored entirely, the state space of the monitor driven composition would not yield any significant improvement. Indeed, during the exploration, when in a symbolic configuration $\langle N, O, s \rangle$, we can either trigger a new non-sensitive event e or trigger a sensitive event e', either new or optional, and change the monitor state. If the first option is chosen, we can also trigger e' from the resulting configuration $\langle N, O \cup \{e\}, s \rangle$, leading to the *same* monitor change. In essence, every possible interleaving of sensitive and non-sensitive events would be explored. This would, of course, be just as inefficient as the traditional explicit algorithm. Fortunately for us, we can do better. As a matter of fact, when in a symbolic configuration $\langle N, O, s \rangle$, we can postpone the sensitive events, and trigger as many non-sensitive events as possible first. This yields a symbolic trace checking algorithm, as presented in Algorithm 6.4. The algorithm works as follows. First, it starts by building a determined monitor for $\neg\varphi$ using Algorithm 6.3 (line 3). Similarly to Algorithm 6.2, the algorithm maintains two sets of symbolic configuration W and Z. Initially, Z is empty and W is filled with the set of initial

```
1  function symbolicLTL-TC(T, φ)
   input  : A po-trace T = ⟨E, V_0, δ, ⪯⟩ and a LTL formula φ
   output : tt if and only if T ⊨_L φ
2  begin
3      ⟨S, s_0, F, 2^P, Δ⟩ := buildDeterminedMonitor(¬φ)
4      Z := ∅, W := ∅
5      forall s ∈ next(s_0, V_0 ∩ prop(φ)) do
6          W := W ∪ {⟨∅, ∅, s⟩}
7      while W ≠ ∅ do
8          Let ⟨N, O, s⟩ ∈ W
9          W := W \ {⟨N, O, s⟩}, Z := Z ∪ {⟨N, O, s⟩}
10         repeat
11             X := O
12             O := O ∪ (enabled(N ∪ O) \ sensitive(s))
13         until X = O
14         if (N ∪ O = E) ∧ (s ∈ F) then
15             returnff
16         forall e ∈ (enabled(N ∪ O) ∪ (O)) ∩ sensitive(s) do
17             forall s' ∈ next(s, e) do
18                 W := (W ∪ {⟨N ∪ ↓e, O \ (↓e ∪ ↑e), s'⟩}) \ Z
19     returntt
20 end
```

Algorithm 6.4 - Symbolic LTL trace checking algorithm

symbolic configurations as specified in Definition 6.2 (lines 4–6). Then, the algorithm loops as long as W contains a symbolic configuration that needs to be examined. At each step of the loop, a symbolic configuration is picked (lines 8–9). This configuration is extended with as many non-sensitive events as possible (lines 10–13). If the resulting symbolic configuration is faulty, i.e. if all events have been treated and the monitor state is final, the algorithm returns false (lines 14–15). If not, the algorithm explores sensitive events according to Definition 6.2 (lines 16–18). Finally, if the exploration stops without finding any faulty configuration, the algorithm returns true (line 19). In order to prove this algorithm is correct, we introduce the covering operator.

Definition 6.5 (Covering Operator)

A symbolic configuration $\langle N_1, O_1, s_1\rangle$ is covered by a symbolic configuration $\langle N_2, O_2, s_2\rangle$, noted $\langle N_1, O_1, s_1\rangle \sqsubseteq_\otimes \langle N_2, O_2, s_2\rangle$, if and only if:

$$(N_2 \subseteq N_1) \wedge (N_1 \cup O_1 \subseteq N_2 \cup O_2) \wedge (s_1 = s_2)$$

Intuitively, a symbolic configuration $\langle N_1, O_1, s_1\rangle$ is covered by another symbolic configuration $\langle N_2, O_2, s_2\rangle$ if $\langle N_2, O_2, s_2\rangle$ represents more explicit configurations. We prove that this operator is monotonic w.r.t. the monitor driven composition.

Lemma 6.3 (Monotonicity of the Covering Operator)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ and a determined monitor $M = \langle S, s_0, F, 2^{\mathbb{P}}, \Delta\rangle$, assuming that $T \otimes M = \langle Q_\otimes, I_\otimes, \rightarrow_\otimes\rangle$, we have that $\forall \langle N_1, O_1, s_1\rangle, \langle N_1', O_1', s_1'\rangle, \langle N_2, O_2, s_2\rangle \in Q_\otimes$:

$$(\langle N_1, O_1, s_1\rangle \rightarrow_\otimes \langle N_1', O_1', s_1'\rangle \wedge \langle N_1, O_1, s_1\rangle \sqsubseteq_\otimes \langle N_2, O_2, s_2\rangle)$$
$$\Rightarrow$$
$$(\exists \langle N_2', O_2', s_2'\rangle \in Q_\otimes : \langle N_2, O_2, s_2\rangle \rightsquigarrow_\otimes \langle N_2', O_2', s_2'\rangle \wedge \langle N_1', O_1', s_1'\rangle \sqsubseteq_\otimes \langle N_2', O_2', s_2'\rangle)$$

Proof

We consider the two cases of Definition 6.4:

(*i*) In the first case, $\exists e \in \mathsf{enabled}(N_1 \cup O_1) \setminus \mathsf{sensitive}(s_1)$ such that $N_1' = N_1$, $O_1' = O_1 \cup \{e\}$ and $s_1' = s_1$. In this case, by Definition 6.5, using a similar argument to the one used in the proof of Lemma 6.2, we can prove that either $e \in O_2$ or $e \in \mathsf{enabled}(N_2 \cup O_2)$. In the former case, we can take $N_2' = N_2$, $O_2' = O_2$ and $s_2' = s_2$. Indeed, we then have that $\langle N_2, O_2, s_2\rangle \rightsquigarrow_\otimes \langle N_2, O_2, s_2\rangle$, $N_2' = N_2 \subseteq N_1 = N_1'$ and $N_1' \cup O_1' = N_1 \cup O_1 \cup \{e\} \subseteq N_2 \cup O_2 = N_2' \cup O_2'$. On the other hand, if $e \in \mathsf{enabled}(N_2 \cup O_2)$, by Definition 6.5, we have that $s_1 = s_2$, which implies that $e \notin \mathsf{sensitive}(s_1) = \mathsf{sensitive}(s_2)$. In this case, we can chose $N_2' = N_2$, $O_2' = O_2 \cup \{e\}$ and $s_2' = s_2$. Indeed, we then have that $\langle N_2, O_2, s_2\rangle \rightsquigarrow_\otimes \langle N_2, O_2 \cup \{e\}, s_2\rangle = \langle N_2', O_2', s_2'\rangle$ by Definition 6.4, $N_2' = N_2 \subseteq N_1 = N_1'$, $N_1' \cup O_1' = N_1 \cup O_1 \cup \{e\} \subseteq N_2 \cup O_2 \cup \{e\} = N_2' \cup O_2'$.

(*ii*) In the second case, $\exists e \in (\mathsf{enabled}(N_1 \cup O_1) \cup O_1) \cap \mathsf{sensitive}(s_1)$ such that $N_1' = N_1 \cap \downarrow e$, $O_1' = O_1 \setminus (\downarrow e \cup \uparrow e)$ and $s_1' \in \mathsf{next}(s_1, e)$. In this case, again, either $e \in O_2$ or $e \in \mathsf{enabled}(N_2 \cup O_2)$. In both instances, we can choose $N_2' = N_2 \cup \downarrow e$, $O_2' = O_2 \setminus (\downarrow e \cup \uparrow e)$ and $s_2' = s_1'$. Indeed, we then have that $\langle N_2, O_2, s_2\rangle \rightsquigarrow_\otimes \langle N_2, O_2 \cup \{e\}, s_2\rangle = \langle N_2', O_2', s_2'\rangle$ by Definition 6.4, $N_2' = N_2 \cup \downarrow e \subseteq N_1 \cup \downarrow e = N_1'$, and $N_2' \cup O_2' = (N_2 \cup \downarrow e) \cup (O_2 \setminus (\downarrow e \cup \uparrow e)) \subseteq (N_1 \cup \downarrow e) \cup (O_1 \setminus (\downarrow e \cup \uparrow e)) = N_1' \cup O_1'$.

Therefore, during the exploration, if a non sensitive event e is triggered from a symbolic configuration $\langle N, O, s\rangle$, since the resulting configuration $\langle N, O\cup\{e\}, s\rangle$ covers the original one, we can discard $\langle N, O, s\rangle$ and only explore $\langle N, O \cup \{e\}, s\rangle$. Using this argument, we can establish the correctness of Algorithm 6.4.

Theorem 6.6 (Correctness of Algorithm 6.4)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ and a LTL formula φ, Algorithm 6.4 returns tt if and only if $T \models_{\mathsf{L}} \varphi$.

Proof

Termination is guaranteed since the number of possible symbolic configurations is finite, and since each configuration is explored only once. We therefore only prove completeness and soundess. In the following, we denote by M the determined monitor built at line 3.

Completeness Assume the algorithm returns true at line 19. It means that no configuration $\langle N, O, s\rangle$ such that $N\cup O = E$ and that $s \in F$ have been encountered in the algorithm. Beware that this does not necessarily mean that such a faulty configuration does not exists in $\mathsf{reach}(T \otimes M)$. Indeed, during the loop at lines 10–13, several configurations are discarded, and therefore not explored. However, at the end of the body of this loop, at line 13, we have that $\langle N, X, s\rangle \sqsubseteq_{\otimes} \langle N, O, s\rangle$. It follows by Lemma 6.3 that if a configuration $\langle N', O', s'\rangle$ is reachable from the discarded configuration $\langle N, X, s\rangle$ then it is covered by a configuration $\langle N'', O'', s''\rangle$ reachable from $\langle N, O, s\rangle$. Moreover, in this case, by Definition 6.5, if $N' \cup O' = E$ and $s' \in F$, we have that $N' \cup O' \subseteq N'' \cup O'' = E$ and that $s'' = s' \in F$. We can therefore conclude that there exists no faulty configuration in $\mathsf{reach}(T \otimes M)$. It follows, by Theorem 6.5, that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) = \varnothing$. Finally, since by construction $\mathsf{lang}(M) = [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$, we have that $T \models \varphi$.

Soundness Assume the algorithm returns false at line 14. Since the algorithm only explores symbolic configurations included in $\mathsf{reach}(T\otimes M)$, it means that there exists a symbolic configuration $\langle N, O, s\rangle \in \mathsf{reach}(T \otimes M)$ such that $N \cup O = E$ and that $s \in F$. It follows by Theorem 6.5 that $\mathsf{traces}(K_T) \cap \mathsf{lang}(M) \neq \varnothing$. By construction, we know that $\mathsf{lang}(M) = [\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$, which directly implies that $T \not\models_{\mathsf{L}} \varphi$.

6.3 Experimental Results

In this section, we compare our new symbolic algorithm from Section 6.2, with the straightforward explicit algorithm from Section 6.1, on randomly generated traces,

Experiment				Explicit			Symbolic		
Model	Proc.	Events	Property	Error	Time	Conf.	Error	Time	Conf.
Peterson	2	10000	φ_1	NO	1.39s	21551	NO	0.35s	4001
	2	100000	φ_1	NO	16.88s	215544	NO	3.45s	40001
	2	1000000	φ_1	↑↑	↑↑	↑↑	NO	34.75s	400002
Peterson	2	10000	φ_1	YES	1.11s	21384	YES	0.01s	4
Faulty	2	100000	φ_1	YES	15.95s	214727	YES	0.05s	4
	2	1000000	φ_1	↑↑	↑↑	↑↑	YES	0.53s	4
ABProtocol	2	10000	φ_2	NO	2.17s	31185	NO	0.42s	4654
	2	100000	φ_2	NO	31.08s	316414	NO	4.25s	46684
	2	1000000	φ_2	↑↑	↑↑	↑↑	NO	43.09s	466887
ABProtocol	2	10000	φ_2	YES	2.06s	31495	YES	0.01s	5
Faulty	2	100000	φ_2	YES	29.70s	315808	YES	0.06s	5
	2	1000000	φ_2	↑↑	↑↑	↑↑	YES	0.53s	5
Philosopher	3	100	φ_3	NO	1.03s	6190	NO	0.05s	299
	5	100	φ_3	NO	87.02s	60727	NO	0.21s	2875
	10	100	φ_3	↑↑	↑↑	↑↑	NO	1.52s	26791
Philosopher	3	100	φ_3	YES	0.09s	1187	YES	0.01s	63
Faulty	5	100	φ_3	YES	78.72s	55982	YES	0.01s	78
	10	100	φ_3	↑↑	↑↑	↑↑	YES	0.01s	55

Figure 6.7 - Experimental results (↑↑ indicates > 1GB)

both in time and in the number of explored configurations. We conducted experiments on several examples of classical distributed systems. For each example, we examined a correct model and a faulty model, where a bug was intentionally introduced. Traces were generated by instrumenting the code using vector clocks, as explained in Chapter 4. We tested both message passing and shared variable programs. Figure 6.7 presents the obtained results. For each experiment, the first four columns respectively present the model, the number of processes, the number of events in the trace and the property. Next, for both the explicit and symbolic method, columns 5 to 10 show if an error was found or not, the time needed for exploration and the number of configurations used. A "↑↑" indicates that no result could be obtained because the process ran out of memory (limited to 1GB for the experiments).

The first example we considered was the *Peterson* mutual exclusion protocol with two processes, where communication is done through shared variables. We tested that mutual exclusion was satisfied. This was done using the LTL formula $\varphi_1 \stackrel{\text{def}}{=} \neg\mathsf{F}(\mathsf{crit}_1 \wedge \mathsf{crit}_2)$ where crit_i models the fact the process i is in his critical section. The second model we considered was the *Alternating-bit protocol* between two process, i.e. a sender and a receiver. This time the communication is achieved using asynchronous message passing. We tested that every message sent was correctly received. This was done using the LTL formula $\varphi_2 \stackrel{\text{def}}{=} \mathsf{G}(\mathsf{send}_0 \Rightarrow \mathsf{F}\,(\mathsf{recv}_0 \vee \mathsf{eot}))$, where send_0, resp. recv_0,

model the sending, respectively reception, of a message tagged with a 0, and where eot models the end of the trace.

On those first two examples, we can already see that the symbolic exploration outperforms the explicit algorithm, both with safety and liveness properties. Also note that, in the faulty version of both models, the error was detected rapidly. This can easily be explained. Indeed, once an error has occurred, the monitor ends up in an accepting state in which no events are sensitive. Therefore, the remaining events of the trace are treated in one single step.

The last example we considered was the *Dining philosophers* problem, where every philosopher takes his left fork first, and then his right fork, except for the first, which takes his right fork first and then his left. We tested that when the first philosopher eats, then his left and right neighbours cannot eat until he finishes. This was done using the LTL formula $\varphi_3 \stackrel{\text{def}}{=} \mathsf{G}(\mathsf{eat}_1 \Rightarrow ((\neg\mathsf{eat}_0 \wedge \neg\mathsf{eat}_2)\,\mathsf{U}\,(\neg\mathsf{eat}_1 \vee \mathsf{eot})))$, where eat_i models the fact that philosopher i is eating. We considered 3, 5 and 10 philosophers. On this example, we can clearly see that using the symbolic algorithm allows to handle a larger number of processes. Indeed, when dealing with 10 philosophers, the explicit exploration fails to terminate, whereas the symbolic algorithm still works.

6.4 Related Works

The problem of analysing partial order traces using temporal logics similar to LTL has already been the subject of numerous works. Directly related to ours is the work of [Sen et al., 2004a], where the authors study the analysis of properties specified using deterministic finite automata, which, as we have seen, is equivalent to LTL. Similarly to what is done using the classical automata based approach, their analysis of the po-trace is done by exploring the lattice of cuts and labelling each cut with states of this automaton. In order to make this technique usable in practice, the authors circumvent the state explosion problem by exploring the lattice of cuts in a level-by-level fashion, and by artificially bounding the maximum number of cuts that are explored at each level. In [Sen et al., 2004b], the authors define a distributed variant of LTL with past modalities called $\mathsf{PT\text{-}DTL}$ (Past Time Distributed Temporal Logic), and present an efficient algorithm for checking that properties expressed in this logic are satisfied. In [Sen et al., 2006], the authors define a logic dedicated to multithreated systems, called $\mathsf{MT\text{-}TL}$ (Multithreated Temporal Logic), also very similar to LTL, and present an efficient algorithm for the detection of properties expressed in this logic. In all those works, the authors chose to restrict the problem either by underapproximating the

state space, or by constraining the logic with ad-hoc modalities, in order to come up with practical algorithms. Alternatively, in this work, we choose to keep the problem intact, and attack the space explosion problem using a symbolic approach.

A lot of research efforts have also been put in the analysis of Mazurkiewicz traces [Mazurkiewicz, 1987] using LTL-like temporal logics. A Mazurkiewicz trace over an alphabet Σ along with an independence relation $I \subseteq \Sigma \times \Sigma$ is a Σ-labelled partial order set of events with special properties not explained here. For such Mazurkiewicz traces, several trace logics, in the same spirit as LTL, have been proposed. Those logics can be categorized as either *local* or *global*. Local trace logics include, amongst others, TR-PTL (Trace Past-time Temporal Logic) introduced in [Thiagarajan, 1994], or TLC (Temporal Logic for Causality) introduced in [Alur et al., 1995]. On the other hand, global trace logics include, amongst others, LTRL (Linear Trace Logic) of [Thiagarajan and Walukiewicz, 2002], or LTL on traces introduced in [Diekert and Gastin, 2002]. Unfortunately, those *trace temporal logics* are not designed to express constraints on the particular order of independent actions. For instance, in a trace, if two actions $a, b \in \Sigma$ are independent, i.e. if $a\, I\, b$, it means that any pair of events labelled respectively with a and b are assumed concurrent. Therefore, a LTL formula like $\mathsf{G}(a \Rightarrow \mathsf{F}\, b)$, expressing that every a is eventually followed by a b is not easily expressible in those *trace logics*. For this reason, we believe that those logics are not well adapted to the testing of distributed systems.

Let us also reference a few works on the definition of monitors for LTL. We already mentioned the work of [Giannakopoulou and Havelund, 2001], adapting the well-known algorithms of [Gerth et al., 1995] for the finite case. We can also mention the work of [Rosu and Havelund, 2005], where the authors use BTT-FSM (Binary Transition Tree - Finite State Machines) instead of finite automata. In [Bauer et al., 2006], the authors define a three-valued semantics of LTL over finite propositional sequences. In this semantics, a LTL formula φ is satisfied (respectively *not* satisfied) by a finite sequence σ if for every infinite extension σ' of σ, φ is satisfied (respectively *not* satisfied) by σ' using the classical infinite semantics. If some extensions satisfy φ, and some extensions do not, the truth value of φ is defined as *unknown*. In this work, the authors also explain how to build a monitor for this semantics, i.e. a finite automaton with both *accepting* and *rejecting* states. Note that the methods defined in this chapter can easily be adapted for this kind of monitors.

Finally, although not directly related to our initial problem, we also have to mention the work of Peled et al. [Peled et al., 2001], where the authors try to improve on classical partial order reduction techniques for the LTL model checking by exploiting

the structure of the monitor obtained using the algorithm of [Gerth et al., 1995]. The main idea is to use the information contained in each node, in particular the *next* field, in order to dynamically change the visibility of the propositions. This allows to significantly enhance the existing algorithms. The underlying ideas in this work are quite similar to ours, although they differ greatly on a technical level.

Chapter 7

CTL Trace Checking

« *The trees that are slow to grow bear the best fruits.* »

Moliere

In Chapter 4, we saw that temporal properties on po-traces could be expressed as temporal logic formulae built on the propositions of the po-trace. This lead us to the definition of the trace checking problem. In Chapter 6, we solved this problem for LTL, by adapting and improving the traditional automata-based model checking algorithm. In this chapter, we tackle the CTL trace checking problem in which it is asked whether the initial global state of a distributed execution satisfies some branching-time temporal requirement expressed as a CTL formula. This problem has been partially solved in [Sen and Garg, 2003], where the authors propose an efficient, i.e. polynomial, algorithm for a subset of CTL, called RCTL. In this subset, only regular temporal predicates can be defined, i.e. CTL formulae whhere satisfying cuts form a lattice. This is a severe restriction however since regular predicates are not closed under disjunction or negation. Moreover, all temporal modalities but EF, EG and AG, are forbidden. This excludes responsiveness questions like, e.g. $\mathsf{AG}(p \Rightarrow \mathsf{AF}(q \vee r))$. Their algorithm is based on computation slicing. Given a RCTL predicate, a slice of the po-trace w.r.t. that predicate is built. The trace checking problem is then solved by checking whether or not the initial empty cut belongs to that slice. In essence, this approach is quite similar to the symbolic approach [McMillan, 1993] traditionally used to solve the CTL model checking problem. Indeed, computational slices can be viewed as a symbolic representation for sets of cuts. Unfortunately, they only allow to represent regular subsets of cuts. In this chapter, in order to tackle full-fledged CTL, we therefore propose an alternative representation for sets of cuts and show how this representation can be exploited to solve efficiently, the trace checking problem.

The remainder of this chapter is structured as follows. First, in Section 7.1, we review the work of [Sen and Garg, 2003] on RCTL and explain how the computation slicing algorithms presented in Chapter 5 can be extended in order to solve the trace checking problem for this subset. Then, in Section 7.2, we explain how the underlying idea behind their approach can be generalised for CTL. For that, we show how sets of cuts can be efficiently represented and manipulated using tuples of natural numbers. Next, in Section 7.3, we examine how those sets of cut can be symbolically represented in practice. Finally, in Section 7.4, we study the efficiency of this symbolic algorithm in practice. This chapter is based on a joint work with Gabriel Kalyon, Thierry Massart and Laurent Van Begin, published in [Kalyon et al., 2007].

7.1 A Regular Fragment of CTL

In Chapter 5, we examined the class of regular predicates, for which the set of satisfying cuts, forming a sub-lattice of the lattice of cuts, could be computed easily using computation slicing and showed how this could be used for the predicate detection problem. This notion of regularity can be easily extended to CTL formulae.

Definition 7.1 (Regular Temporal Predicates)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, a CTL formula φ is regular w.r.t. T if and only if:

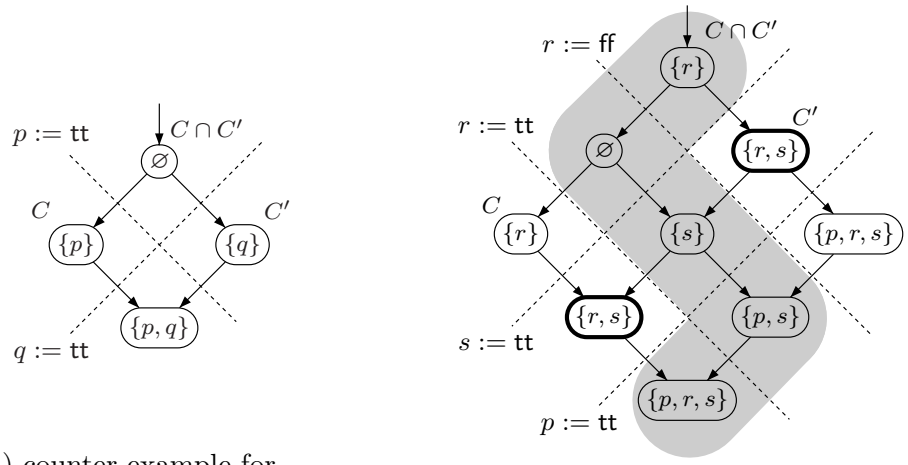
$$\forall C, D \in Q : \left(\langle T, C\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi \wedge \langle T, D\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi\right) \Rightarrow \left(\langle T, C \cap D\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi \wedge \langle T, C \cup D\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi\right)$$

Regular temporal predicates have been studied in [Sen and Garg, 2003]. In particular, the authors studied how the application of CTL temporal modalities affects the regularity of predicates. They showed that regular temporal predicates are closed under the temporal modalities EF, AG and EG.

Theorem 7.1 (Regular-Preserving CTL Modalities [Sen and Garg, 2003])

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and CTL formula φ, we have that if φ is regular w.r.t. T, then so are $\mathsf{EF}\,\varphi$, $\mathsf{AG}\,\varphi$ and $\mathsf{EG}\,\varphi$.

Furthermore, they showed that for any other CTL modality, this is not the case, i.e. temporal regular predicates are not closed under EX, AX, EU, AU and AF. Counter-examples for each of those modalities, inspired from [Sen and Garg, 2003], are presented hereafter.



(a) counter-example for EX, AX, EU and AU

(b) counter-example for AF

Figure 7.1 - Counter-examples for the regularity of regular temporal modalities

Example 7.1

Figure 7.1 gives counter-examples showing that regular temporal predicates are not closed to EX, AX, EU, AU and AF. First, Figure 7.1 (a) shows a counter-example for EX, AX, EU and AU. In this example, the predicates p, q and $p \wedge q$ are regular. However, $\mathsf{EX}(p \wedge q)$, $\mathsf{AX}(p \wedge q)$, $\mathsf{E}(p \,\mathsf{U}\, q)$ and $\mathsf{A}(p \,\mathsf{U}\, q)$ are not, since they all hold in the cuts C and C', but not in the cut $C \cap C'$. Figure 7.1 (b) shows the counter-example for AF. In this lattice, the predicate $r \wedge s \wedge \neg p$ is regular. The satisfying cuts are highlighted in bold. We can clearly see that both cuts C and C' satisfy $\mathsf{AF}\,(r \wedge s \wedge \neg p)$. However, the cut $C \cap C'$ above does not, because of the trace highlighted in gray.

Based on this observation, the authors define a fragment of CTL, called RCTL, in which all predicates are regular. This fragment is obtained by keeping only regular-preserving operations, i.e. conjunction, EF, AG and EG. The formal definition follows.

Definition 7.2 (Syntax of RCTL [Sen and Garg, 2003])

Given a set of propositions $\mathbb{P}$, a formula in regular computational tree logic (RCTL) is defined using the following grammar:

$$\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi \wedge \varphi \mid \mathsf{EF}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{AG}\,\varphi$$

where $p \in \mathbb{P}$.

The authors then extend the slicing algorithm presented in Chapter 5 to take into account the regular modalities, thus allowing them to obtain an efficient RCTL trace checking algorithm. In the remainder of this section, we explain this approach in more details. In Sections 7.1.1 to 7.1.3, we present and illustrate the slicing algorithms for the three RCTL temporal modalities EF, AG and EG. We then explain, in Section 7.1.4, how this can be used to obtain a trace checking algorithm for RCTL predicates.

7.1.1 Slicing Algorithm for EF

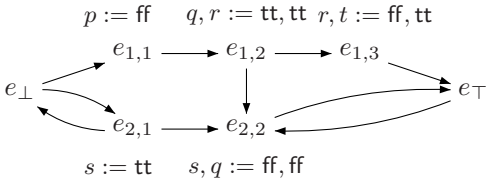
For formulae of the form $\mathsf{EF}\,\varphi$, the slice is built from the slice of $\top$ (representing the entire set of cuts) using the slice of φ as follows. By definition, a cut C satisfies $\mathsf{EF}\,\varphi$ if and only if there exists a cut D reachable from C where φ holds. However, since φ is regular we know that the set of cuts satisfying φ is a lattice w.r.t. to set inclusion, and therefore has a unique maximal element. It follows that a cut C satisfies $\mathsf{EF}\,\varphi$ if and only if this maximal cut is reachable from C. The slice of a po-trace w.r.t. $\mathsf{EF}\,\varphi$ is consequently the slice representing exactly those cuts. To compute this slice, we must first obtain the maximal cut that satisfies φ. This is simply the last consistent cut in the slice of the trace w.r.t. φ. The slice of $\mathsf{EF}\,\varphi$ is then built from the slice of $\top$, by removing any cut from which this last consistent cut is not reachable. In practice, as illustrated in Example 7.2, we start from the slice of $\top$, to which are added all the edges in the strongly connected component of $e_\top$ in the slice of φ.

Example 7.2

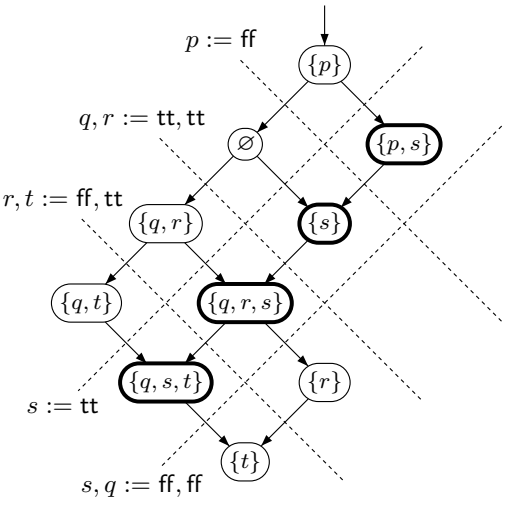
Figure 7.2 illustrates the computation of the slice for a formula of the form $\mathsf{EF}\,\varphi$. First, Figure 7.2(a) presents the slice of the example po-trace of Chapter 4, w.r.t. the formula s. Figure 7.2(b) shows the corresponding consistent cuts. The last consistent cut in this slice is $C \stackrel{\text{def}}{=} \{e_{1,1}, e_{1,2}, e_{1,3}, e_{2,1}\}$. This is the cut labelled with $\{q, s, t\}$ in Figure 7.2(b). Next, Figure 7.2(c) shows the slice of that same trace w.r.t. $\mathsf{EF}\,s$ computed as explained above. Indeed, consider the strongly connected component of $e_\top$ in the slice of s. This component contains two edges $\langle e_{2,2}, e_\top\rangle$ and $\langle e_\top, e_{2,2}\rangle$, which are therefore added to the slice of $\top$ to obtain the slice of $\mathsf{EF}\,s$. The corresponding consistent cuts are presented in Figure 7.2(d). Note that this is exactly those cuts that are included in C, thus satisfying $\mathsf{EF}\,s$.

7.1.2 Slicing Algorithm for AG

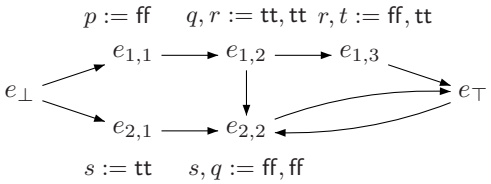
For formulae of the form $\mathsf{AG}\,\varphi$, the slice is built from the slice of φ by adding edges to remove from this slice those cuts that do not satisfy $\mathsf{AG}\,\varphi$. By definition, a cut C



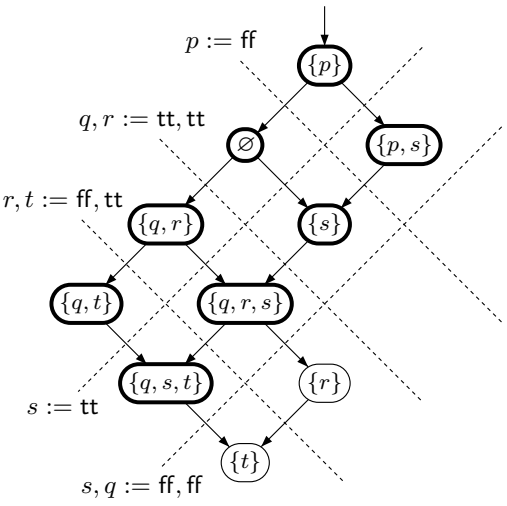
(a) a slice of T w.r.t. s



(b) cuts of T consistent with (a)



(c) slice of T w.r.t. $\mathsf{EF}\, s$



(d) cuts of T consistent with (c)

Figure 7.2 - Computation slicing for the EF modality

does not satisfy $\mathsf{AG}\,\varphi$ if and only if there exists a cut D reachable from C that does not satisfy φ. Since D does not satisfy φ, it is not consistent with the slice of φ, which means that there is an edge $\langle e, e' \rangle$ in this slice such that e' belongs to D but not e. Note that this edge was added specially for φ, since D is a valid cut. Based on this observation, the authors of [Sen and Garg, 2003] proved that a cut C satisfies $\mathsf{AG}\,p$ if and only if it includes the node e of each additional vertex $\langle e, e' \rangle$ in the slice of φ, i.e. edges that do not belong to the slice of $\top$. In practice, as illustrated in Example 7.3, in order to build this slice, an edge from the node e to the node $e_\perp$ is added for each additional vertex $\langle e, e' \rangle$ in the slice of T w.r.t. φ.

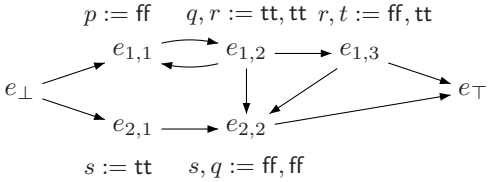
Example 7.3

Figure 7.3 illustrates the computation of the slice of a formula of the form $\mathsf{AG}\,\varphi$. First, Figure 7.3(a) presents the slice of the same po-trace as in Example 7.2, w.r.t. the regular predicate $p \vee q \vee t$ [(1)]. Figure 7.3(b) shows the corresponding set of consistent cuts. In this slice, two edges were added from the slice of $\top$: $\langle e_{1,2}, e_{1,1} \rangle$ and $\langle e_{1,3}, e_{2,2} \rangle$. Therefore, in order to build the slice for $\mathsf{AG}(p \vee q \vee t)$, we must add to this slice two edges $\langle e_{1,2}, e_\perp \rangle$ and $\langle e_{1,3}, e_\perp \rangle$. The resulting slice is presented in Figure 7.3(c). The corresponding set of consistent cuts is presented in Figure 7.3(d). Note that those are exactly the cuts satisfying $\mathsf{AG}(p \vee q \vee t)$.

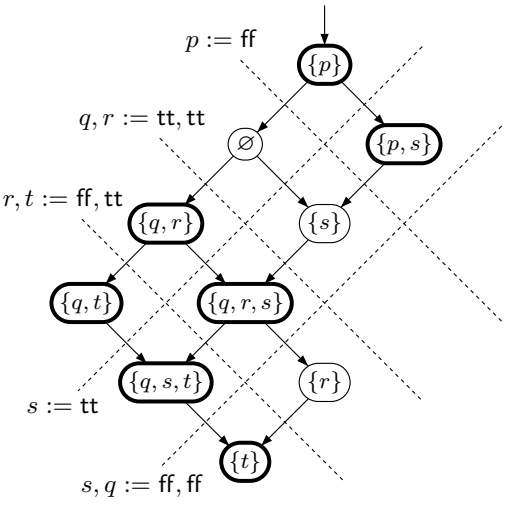
7.1.3 Slicing Algorithm for EG

For formulae of the form $\mathsf{EG}\,\varphi$, the slice is also built from the slice of φ by adding edges to remove those cuts that do not satisfy $\mathsf{EG}\,\varphi$. By definition, a cut C does not satisfy $\mathsf{EG}\,\varphi$ if and only if for every run of T that starts in C, there exists a cut D in this run where φ does not hold, or in other words if violating φ is unavoidable. Based on this observation, the authors of [Sen and Garg, 2003] proved that this is the case if and only if there exists a non trivial (i.e. of size greater than 1) strongly connected component in the slice of φ that is not included in C. Indeed, whichever run is chosen from C, this run will have to reach a cut D that splits this strongly connected component, and in doing so will violate φ. Therefore, the slice of T w.r.t. $\mathsf{EG}\,\varphi$ is built from the slice of φ by removing the cuts that do not contain every non trivial strongly connected component contained in the slice of φ. In practice, as illustrated in Example 7.4, this is done by adding an edge from the node e to the node $e_\perp$ for each additional vertex $\langle e, e' \rangle$ in the slice of φ that creates a non-trivial strongly connected component.

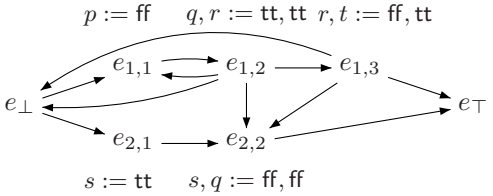
[1] note that in general, regular predicates are not closed to disjunction; in this case, the predicate $p \vee q \vee t$ is regular because of the particular structure of the lattice of cuts.



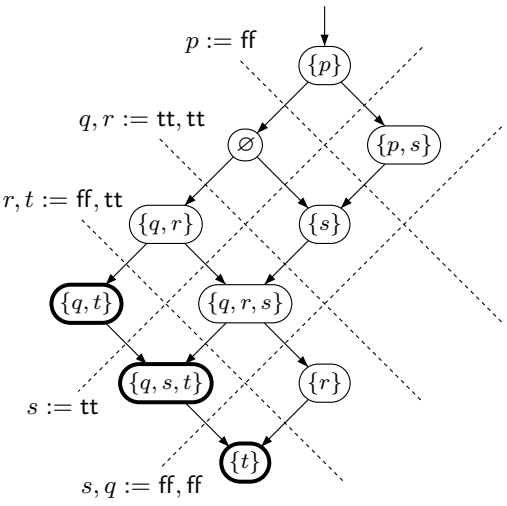
(a) a slice of T w.r.t. $p \vee q \vee t$



(b) cuts of T consistent with (a)

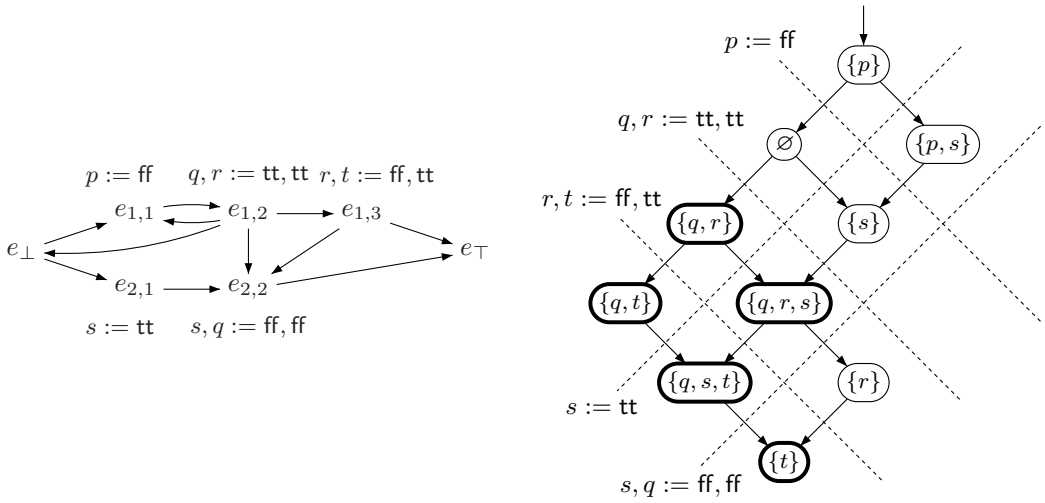


(c) slice of T w.r.t. $\mathsf{AG}(p \vee q \vee t)$



(d) cuts of T consistent with (c)

Figure 7.3 - Computation slicing for the AG modality



(a) slice of T w.r.t. $\mathsf{EG}(p \vee q \vee t)$ (b) cuts of T consistent with (b)

Figure 7.4 - Computation slicing for the EG modality

Example 7.4

Figure 7.4 illustrate the computation of the slice of a formula of the form $\mathsf{EG}\,\varphi$. Figure 7.4(a), shows the slice of the same trace as in the previous two examples w.r.t. $\mathsf{EG}(p \vee q \vee t)$. This slice is built from the slice of $p \vee q \vee t$, presented in Figure 7.3(a), by considering the non trivial strongly connected component. In this case, there is only one component created by the additional edge $\langle e_{1,2}, e_{1,1}\rangle$. Therefore, in order to build the slice for $\mathsf{EG}(p \vee q \vee t)$, an edge $\langle e_{1,2}, e_{\perp}\rangle$ is added. The corresponding consistent cuts are depicted in Figure 7.4(b). Note that those are exactly the cuts satisfy $\mathsf{EG}(p \vee q \vee t)$.

7.1.4 Trace Checking Algorithm

Using the slicing techniques presented above, the authors of [Sen and Garg, 2003] define an efficient trace checking algorithm, which is formalized in Algorithm 7.1. This algorithm works as follows. First, it computes the slice of the po-trace w.r.t. the given formula φ. For basic non-temporal constructs like $\top$, $\perp$, literals and conjunction, the algorithm proceeds in the same way as Algorithm 5.8 (lines 3–6) from Chapter 5. For temporal modalities, the algorithm builds the slice as explained in Sections 7.1.1 to 7.1.3 (lines 7–21). Finally, once the slice is computed, the algorithm checks that the initial cut belongs to it. It can be easily proven that this is the case if and only if the node $e_{\perp}$ has no incoming edges (line 27).

```
1  function computeSliceRCTL(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a RCTL predicate φ
   returns: R_φ ⊆ N × N such that ⟨N, R_φ⟩ is a slice of T w.r.t φ
2  begin
3      if (φ = ⊤) ∨ (φ = ⊥) ∨ (φ = p) ∨ (φ = ¬p) then
4          return computeSlice(T, φ)
5      else if φ = φ_1 ∧ φ_2 then
6          R_φ := computeSliceRCTL(T, φ_1) ∪ computeSliceRCTL(T, φ_2)
7      else if φ = EF φ_1 then
8          R_{φ_1} := computeSliceRCTL(T, φ_1), R_⊤ := computeSlice(T, ⊤)
9          R_φ = R_⊤, X := connectedComponent(⟨N, R_{φ_1}⟩, e_⊤)
10         forall ⟨e, e'⟩ ∈ R_{φ_1} \ R_⊤ s.t. {e, e'} ⊆ X do
11             R_φ := R_φ ∪ ⟨e_⊤, e⟩
12     else if φ = AG φ_1 then
13         R_{φ_1} := computeSliceRCTL(T, φ_1)
14         R_⊤ := computeSlice(T, ⊤), R_φ = R_{φ_1}
15         forall ⟨e, e'⟩ ∈ R_{φ_1} \ R_⊤ do
16             R_φ := R_φ ∪ {⟨e, e_⊥⟩}
17     else if φ = EG φ_1 then
18         R_{φ_1} := computeSliceRCTL(T, φ_1)
19         R_⊤ := computeSlice(T, ⊤), R_φ = R_{φ_1}
20         forall ⟨e, e'⟩ ∈ R_{φ_1} \ R_⊤ s.t. |connectedComponent(⟨N, R_{φ_1}⟩, e)| > 1 do
21             R_φ := R_φ ∪ {⟨e, e_⊥⟩}
22     return R_φ
23 end

24 function slicingBasedRCTL-TC(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩, a RCTL predicate φ
   returns: tt if and only T ⊨_c φ
25 begin
26     N = E ∪ {e_⊤, e_⊥}, R_φ := computeSliceRCTL(T, φ)
27     return (∃e ∈ N : ⟨e, e_⊥⟩ ∈ R_φ)
28 end
```

Algorithm 7.1 - RCTL trace checking algorithm

7.2 Tuple Based Approach

Slices can be viewed, in some sense, as a symbolic data structure for representing and manipulating sets of cuts. From this point of view, the approach proposed in [Sen and Garg, 2003] is very similar to the classical symbolic approach to CTL model checking [McMillan, 1993], where the set of states satisfying a given CTL formula is inductively computed using efficient symbolic data structures. This idea could be easily generalised to handle any CTL formula. However, slices are too restrictive for that, since they only allow to represent regular sets of cuts. If we want to solve the trace checking problem for full-fledged CTL, we need a way to represent and manipulate arbitrary, in particular non-regular, sets of cuts. Our proposal is based on the following observation. By definition, in a partial order trace, the events of each process are totally ordered. Therefore, in order to represent a cut, we can simply remember, for each process, the number of events of that process that have already been triggered in this cut. With this idea, cuts and therefore sets of cuts can be effectively represented using k-uples of natural numbers. Then, similarly to what is done for RCTL, given a po-trace T and a CTL formula φ, we can solve the trace checking problem by first computing the set of tuple corresponding to $[\![\varphi]\!]_{\mathsf{C}}^{T}$, and then checking whether the tuple corresponding to the initial empty cut belongs to this set. Hence, given a po-trace T and CTL formula φ, all we need is a way to compute the set of tuples corresponding to $[\![\varphi]\!]_{\mathsf{C}}^{T}$. The main advantage of this method is that, as will be shown in Section 7.3, there exists a number of efficient symbolic data structures that can be used for representing and manipulating sets of tuple of naturals.

In the remainder of this section, we explain how the set of tuples corresponding to $[\![\varphi]\!]_{\mathsf{C}}^{T}$ can be computed inductively on the structure of φ. We start in Section 7.2.2 and Section 7.2.3 with the initial cases, i.e. when $\varphi = \top$ or $\varphi = p$. Then, in Section 7.2.4, we examine how to handle boolean operators. Finally, temporal modalities are taken care of in Section 7.2.5. But before that, in Section 7.2.1, we first examine in more detail how cuts can be represented as tuples of natural numbers.

7.2.1 Tuple Representation

Condition (i) in Definition 4.3 of partial order traces implies that the events of each process of the trace are totally ordered. Therefore, for a cut C, one can simply remember the number of events of process P_i that have already been executed in C. More precisely, a cut C can be represented by a k-uple of naturals where the i^{th} component gives the number of events of process P_i that already occurred in C. The formal

definition follows.

Definition 7.3 (Tuple Representation of Cuts)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a cut $C \in Q$, the tuple representation of C, noted $\mathsf{tuple}(C)$, is a tuple $t \in \mathbb{N}^k$ such that:

$$\forall i \in [1, k] : t[i] = |C \cap P_i|$$

Using this representation, testing if a cut is included in another can then simply be done by a component-wise comparison of their respective tuple representation.

Proposition 7.1

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and two cuts $C, D \in Q$, we have that:

$$(C \subseteq D) \Leftrightarrow (\mathsf{tuple}(C) \leq \mathsf{tuple}(D))$$

The union, respectively intersection, can also be computed by taking the componentwise maximum, respectively minimum.

Proposition 7.2

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and two cuts $C, D \in Q$, we have that:

(i) $\forall i \in [1, k] : \mathsf{tuple}(C \cup D)[i] = \mathsf{max}_{\leq}(\mathsf{tuple}(C)[i], \mathsf{tuple}(D)[i])$

(ii) $\forall i \in [1, k] : \mathsf{tuple}(C \cap D)[i] = \mathsf{min}_{\leq}(\mathsf{tuple}(C)[i], \mathsf{tuple}(D)[i])$

Furthermore, checking whether an event e belongs to a cut can be done easily on the tuple representation. Indeed, it is necessary and sufficient to check that the process P_i to which e belongs contains at least all the predecessors of e in P_i.

Proposition 7.3

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a cut $C \in Q$, we have that:

$$\forall i \in [1, k], \forall e \in P_i : (e \in C) \Leftrightarrow (\mathsf{tuple}(C)[i] \geq |P_i \cap \downarrow e|)$$

The tuple representation can be naturally extended to sets of cuts. Given a set $X \subseteq Q$, $\mathsf{tuple}(X) \stackrel{\mathbf{def}}{=} \{\mathsf{tuple}(C) \mid C \in X\}$. Using this notation, given a po-trace T and a CTL formula φ, the set of tuples corresponding to $[\![\varphi]\!]^T_{\mathsf{C}}$, noted $[\![\varphi]\!]^T_{\mathbb{N}^k}$, is defined as $\mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}})$.

```
1 function computeTuplesTautology(T)
  input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes
  returns: ⟦⊤⟧^T_{ℕ^k}
2 begin
3     R := env(T)
4     H := {⟨e, e'⟩ ∈ ⪯| (e ≠ e') ∧ (∄e'' ∈ E \ {e, e'} : e ⪯ e'' ⪯ e')}
5     forall ⟨e, e'⟩ ∈ H s.t. ∃i, j ∈ [1, k] : (e ∈ P_i) ∧ (e' ∈ P_j) ∧ (i ≠ j) do
6         R := R \ (after(e') ∩ before(e))
7     return R
8 end
```

Algorithm 7.2 - Computation of the set of tuples satisfying $\top$

7.2.2 Tautology

If $\phi \equiv \top$, we need to compute the set of tuples representing all possible cuts of the trace $T = \langle E, V_0, \delta, \preceq \rangle$. Our construction is based on the Hasse diagram $\langle E, H \rangle$ of T. We start as if this diagram had no communication edge, i.e. no edge $\langle e, e' \rangle \in H$ where e and e' belong to different processes. The corresponding set of tuples is then simply the set of all tuples in the bounds imposed by the trace. We call this set of tuples the *envelope* of T. Formally, given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes, the envelope of T, is defined as $\mathsf{env}(T) \stackrel{\text{def}}{=} \{t \in \mathbb{N}^k \mid \forall i \in [1, k] : t[i] \leq |P_i|\}$. Then, we take the communication edges into account one at a time. For that purpose, given such a communication edge $\langle e, e' \rangle$, we need to remove the tuples corresponding to the sets of events that are not downward closed because of that edge, i.e. the sets of events containing e', but not e. For that, we can use Proposition 7.3, which states that an event $e \in P_i$ belongs to a cut C if and only if $\mathsf{tuple}(C)[i] \geq |P_i \cap \downarrow e|$. Given an event $e \in P_i$ for some $i \in [1, k]$, let us note $\mathsf{after}(e) \stackrel{\text{def}}{=} \{t \in \mathsf{env}(T) \mid t[i] \geq |P_i \cap \downarrow e|\}$ and $\mathsf{before}(e) \stackrel{\text{def}}{=} \{t \in \mathsf{env}(T) \mid t[i] < |P_i \cap \downarrow e|\}$. The set of tuples corresponding to the cuts forbidden by a communication edge $\langle e, e' \rangle$ is then simply given by $\mathsf{after}(e') \cap \mathsf{before}(e)$. This construction is formalized in Algorithm 7.2, which works as follows. First, the set R, which will be returned at the end, is initialised with the envelope of T (line 3). Then, the edges of the Hasse diagram are collected in a set H (line 4). Next, for each edge $\langle e, e' \rangle \in H$ such that e and e' belong to different processes, the tuples forbidden by this edge are removed from R (line 6). Finally, once all communication edges have been taken into account, the algorithm returns R (line 7).

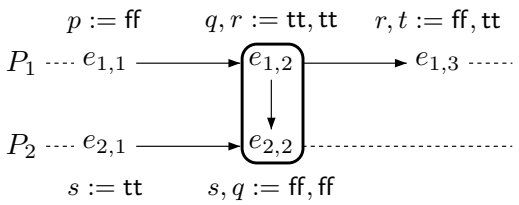


Figure 7.5 - Computation of the set of tuples satisfying $\top$

Example 7.5

Figure 7.5 illustrates the construction of the set of tuples corresponding to the set of all cuts of the po-trace T from Figure 4.4 (page 95), namely $[\![\top]\!]^T_{\mathbb{N}^k}$. In this example, there is only one communication edge, i.e. $\langle e_{1,2}, e_{2,2}\rangle$. The set of tuples is therefore given by $[\![\top]\!]^T_{\mathbb{N}^k} = \mathsf{env}(T) \setminus \mathsf{after}(e_{2,2}) \cap \mathsf{before}(e_{1,2})$.

The correctness of this algorithm is proven as follows. First, we prove that the algorithm is sound, i.e. set of tuples returned by Algorithm 7.2 is included in $\mathsf{tuple}(Q)$.

Lemma 7.1 (Soundness of Algorithm 7.2)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ of k processes such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$, we have that if Algorithm 7.2 terminates, then at the end $R \subseteq \mathsf{tuple}(Q)$.

Proof

Consider a tuple $t \in R$ at the end of the algorithm. Let us build a set of events $F = \bigcup_{i\in[1,k]} \{e \in P_i \mid |P_i \cap \downarrow e| \leq t[i])\}$. We prove by contradiction that $F \in Q$. Indeed assume $F \notin Q$. In this case, we know that there exists two events $e \preceq e'$ with $e' \in F$ and $e \notin F$. Since $e \preceq e'$, we know that there exists a sequence of event $e = e_1 e_2 \dots e_\ell = e'$ connecting e to e' in the Hasse diagram of T, i.e. $\forall k \in [1, l) : \langle e_k, e_{k+1}\rangle \in H$. Moreover, by construction of F, it must be that e and e' belong to separate processes. It follows that the sequence contains at least one pair $\langle e_k, e_{k+1}\rangle$ such that e_k and e_{k+1} belong to separate process. However, in this case, t would have been removed from R at line 6 because of that edge. It follows that $F \in Q$. Finally, from the construction of F, we have that $\mathsf{tuple}(F) = t$. Hence, we conclude.

Then, we prove its completeness, i.e. that any tuple in $\mathsf{tuple}(Q)$ is included in the set of tuples returned by Algorithm 7.2.

Lemma 7.2 (Completeness of Algorithm 7.2)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, we have that if Algorithm 7.2 terminates then at the end $\mathsf{tuple}(Q) \subseteq R$.

Proof

First, by Definition 7.3, for any cut $C \in Q$, we have that $\forall i \in [1, k] : 0 \leq \mathsf{tuple}(C)[i] \leq |P_i|$. It follows that at the beginning of the algorithm (line 3), $\forall C \in Q : \mathsf{tuple}(C) \in R$, or in other words that $\mathsf{tuple}(Q) \subseteq R$. Then, we show that this is an invariant of the loop at lines 5–6. We proceed by contradiction. Indeed consider a tuple $t \in \mathsf{after}(e') \cap \mathsf{before}(e)$ for some communication edge $\langle e, e' \rangle$, and assume the existence of a cut $C \in Q$ such that $\mathsf{tuple}(C) = t$. By construction of C, we have that $e' \in C$ and that $e \notin C$. However, by construction of H, we know that $e \preceq e'$. It follows that $C \notin \mathsf{DC}_{\preceq}(E) = Q$, which contradicts our initial assumption. It follows that $(\mathsf{after}(e') \cap \mathsf{before}(e)) \cap \mathsf{tuple}(Q) = \varnothing$. Hence we conclude.

Finally, this allows us to establish its correctness.

Theorem 7.2 (Correctness of Algorithm 7.2)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ of k processes such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, Algorithm 7.2 returns $[\![\top]\!]^T_{\mathbb{N}^k}$.

Proof

We successively prove partial correctness and termination.

Partial Correctness Thanks to Lemma 7.1 and Lemma 7.2, we can conclude that if the algorithm terminates, then $R = \mathsf{tuple}(Q)$. Then, since $[\![\top]\!]^T_{\mathbb{N}^k} = \{t \in \mathbb{N}^k \mid \exists C \in Q : (\mathsf{tuple}(C) = t) \wedge \langle T, C \rangle \models^{\intercal}_{\mathsf{C}} \top)\} = \mathsf{tuple}(Q)$, we conclude.

Termination By definition, T contains only a finite number of events, which implies that $|H|$ computed at line 4 is finite. Then, since the loop at lines 5–6 will execute at most $|H|$ times, we conclude.

7.2.3 Propositions

If $\phi \equiv p$ for some $p \in \mathbb{P}$, we proceed as follows. By condition (ii) of Definition 4.3, we know that the set of events that modify the truth value of p, namely $E_{/p}$, is totally ordered. Amongst those events, some set p to true and some set p to false. In order to compute $[\![p]\!]^T_{\mathbb{N}^k}$, we simply compute the union of all chunks of cuts in the trace where p has been set to true, but not yet back to false. In practice, as illustrated in Example 7.6, for each event e that sets p to true, if e' is the first event after e that sets p back to false, we add the set of tuples after e and before e' that correspond to

```
1  function computeTuplesPropositions(T, p)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes and a propositions p ∈ ℙ
   returns: [[p]]^T_{ℕ^k}
2  begin
3      U := {e ∈ E_{/p} | p ∈ δ(e, tt)}, D = E_{/p} \ U, R = ∅
4      if p ∈ V_0 then
5          S := env(T)
6          if D ≠ ∅ then
7              S := S ∩ before(min_⪯(D))
8          R := R ∪ S
9      forall e ∈ U do
10         S := after(e)
11         if D ∩ ↑e ≠ ∅ then
12             S := S ∩ before(min_⪯(D ∩ ↑e))
13         R := R ∪ S
14     R := R ∩ computeTuplesTautology(T)
15     return R
16 end
```

Algorithm 7.3 - Computation of the set of tuples satisfying a proposition

actual cuts to the result, i.e. $(\mathsf{after}(e) \cap \mathsf{before}(e')) \cap [\![\top]\!]^T_{\mathbb{N}^k}$. Of course, on top of that, special precautions must be taken at the beginning and at the end of the trace, i.e. if p already holds at the beginning of the trace and/or if the last event of $E_{/p}$ is one that sets p to true. This construction is formalized in Algorithm 7.3. We start with three sets of events. The first set U contains the events that set p to true, the second set D contains those that set p to false and the last set R, initially empty, is the set of tuples that will be returned at the end (line 3). If p holds at the beginning of the trace (line 4), the algorithm adds to R the set of tuples from the beginning of the trace in which p has not been set back to ff. For that purpose, the algorithm starts with a set S initially containing all tuples in the envelope of T (line 5). Then, S is restricted to all tuples before the first event that sets p to false, if such an event exists (lines 6–7). Next, the content of S is added to R (line 8). Then, for each event e that sets p to true (line 9), the algorithm adds to R the set of tuples after e in which p is not set back to false yet. For that purpose, the algorithm starts with a set S initially containing

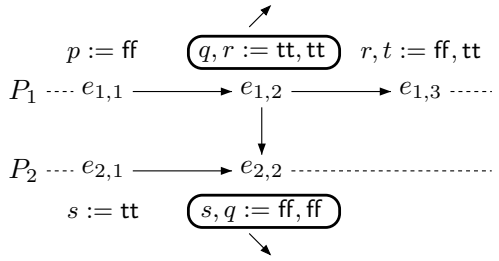


Figure 7.6 - Computation of the set tuples satisfying a proposition

the set of all tuples after e (line 10). Then, S is restricted to all tuples before the first event after e that sets p to false, if such an event exists (lines 11–12). Next, the content of S is added to R (line 13). Finally, once all the events in U have been considered, R is restricted to tuples representing actual cuts (line 14) and returned (line 15). In particular, if p is true all along the po-trace, i.e. if $p \in V_0$ and $D = \varnothing$, R is initialized to $\mathsf{env}(T)$. Therefore, since the main loop only adds tuple to R, the algorithm returns $\mathsf{env}(T) \cap [\![\top]\!]^T_{\mathbb{N}^k} = [\![\top]\!]^T_{\mathbb{N}^k}$, i.e. the set of all cuts. Symmetrically, if p is false all along the po-trace, i.e. if $p \notin V_0$ and $U = \varnothing$, R is initialized to $\varnothing$, the main loop the loop at is not executed (since $U = \varnothing$) and the algorithm returns $\varnothing$.

Example 7.6

Figure 7.6 illustrates the construction of $[\![q]\!]^T_{\mathbb{N}^k}$ on the example partial order trace from Chapter 4. In this example, there are only two events dealing with q, i.e. $E_{/q} = \{e_{1,2}, e_{2,2}\}$. The first of those two events sets q to true, while the second one sets q to false. Therefore, the set of tuples corresponding to cuts where q holds is given by $[\![q]\!]^T_{\mathbb{N}^k} = (\mathsf{after}(e_{1,2}) \cap \mathsf{before}(e_{2,2})) \cap [\![\top]\!]^T_{\mathbb{N}^k}$.

The correctness of Algorithm 7.3 is proven as follows. First, we prove soundness, i.e that p holds in all the cuts returned by the algorithm.

Lemma 7.3 (Soundness of Algorithm 7.3)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over a set of propositions $\mathbb{P}$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a proposition $p \in \mathbb{P}$, we have that if Algorithm 7.3 terminates, then at the end $R \subseteq [\![\top]\!]^T_{\mathbb{N}^k}$.

Proof

Consider a tuple $t \in R$ at the end of the algorithm. By construction (line 14) and by Theorem 7.2, we have that $R \subseteq [\![\top]\!]^T_{\mathbb{N}^k}$, and thus that there exists a cut $C \in Q$ such that $\mathsf{tuple}(C) = t$. From there, we consider two cases:

Proof (cont'd)

(i) If t was added before the main loop at line 8 then, by construction, we have that $C \cap D = \varnothing$. In other words C does not contain any event setting p to false. It follows that $p \in \mathcal{L}(C)$ if and only if $p \in V_0$. However, because of the test at line 4, we know that $p \in V_0$, and therefore that $p \in \mathcal{L}(C)$.

(ii) If t was added during the loop at line 13 for some event $e \in U$, then by construction, we have that $t \in \mathsf{after}(e)$, which implies by Proposition 7.3, that $e \in C$. Moreover, for any event $e' \in D$ such that $e \preceq e'$, we have that $e' \notin C$. In other words, after e, C contains no event that sets p to false. Since $e \in U$, we can therefore conclude that $p \in \delta(\mathsf{max}_{\preceq}(C_{/p}), \mathsf{tt})$.

In both cases, by Definition 4.4, we have that $\langle T, C\rangle \models^{\mathsf{T}}_{\mathsf{C}} p$ and thus that $t \in [\![p]\!]^T_{\mathbb{N}^k}$.

Then, we prove its completeness, i.e. that any tuple representing a cut that satisfies p is included in R.

Lemma 7.4 (Completeness of Algorithm 7.3)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$ over a set of propositions $\mathbb{P}$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow\rangle$ and a proposition $p \in \mathbb{P}$, we have that if Algorithm 7.3 terminates, then at the end $[\![p]\!]^T_{\mathbb{N}^k} \subseteq R$.

Proof

Consider a cut $C \in [\![p]\!]^T_{\mathsf{C}}$. By Definition 4.4, either $C_{/p} = \varnothing$, in which case $p \in V_0$, or $C_{/p} \neq \varnothing$, in which case $p \in \delta(\mathsf{max}_{\preceq}(C_{/p}), \mathsf{tt})$. We examine these two possibilities separately:

(i) If $C_{/p} = \varnothing$, since $p \in V_0$, the test at line 4 will succeed. Therefore, since $\mathsf{tuple}(C) \in \mathsf{env}(T)$, we have that $\mathsf{tuple}(C) \in S$ at line 5. Then, if $D \neq \varnothing$, since $C \cap D = \varnothing$, we have in particular that $\mathsf{min}_{\preceq}(D) \notin C$. It follows, by Proposition 7.3, that $\mathsf{tuple}(C) \in \mathsf{before}(\mathsf{min}_{\preceq}(D))$, which implies that C is kept in S at line 7. We can therefore conclude that $\mathsf{tuple}(C)$ is added to R at line 8.

(ii) If $C_{/p} \neq \varnothing$, we have that $p \in \delta(\mathsf{max}_{\preceq}(C_{/p}, \mathsf{tt}))$. In this case, since $e \stackrel{\mathbf{def}}{=} \mathsf{max}_{\preceq}(C_{/p}) \in U$, and since $\mathsf{tuple}(C) \in \mathsf{after}(e)$, we have that $\mathsf{tuple}(C) \in S$ at line 10. Then, if $D \cap \uparrow e \neq \varnothing$, since $C \cap (D \cap \uparrow e) = \varnothing$, we have in particular that $\mathsf{min}_{\preceq}(D \cap \uparrow e) \notin C$. It follows, by Proposition 7.3, that $\mathsf{tuple}(C) \in \mathsf{before}(\mathsf{min}_{\preceq}(D \cap \uparrow e))$, which implies that C is kept in S at line 12. We can therefore conclude that $\mathsf{tuple}(C)$ is added to R at line 13.

In both cases, $C \in R$. It follows directly that $[\![p]\!]^T_{\mathbb{N}^k} \subseteq R$.

Finally, this allows us to establish its correctness.

Theorem 7.3 (Correctness of Algorithm 7.3)

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ over a set of propositions $\mathbb{P}$ such that $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$ and a proposition $p \in \mathbb{P}$, Algorithm 7.3 returns $[\![p]\!]^T_{\mathbb{N}^k}$.

Proof

We successively prove partial correctness and termination

Partial Correctness Thanks to Lemma 7.3 and Lemma 7.4, we have that if the algorithm terminates, then $R = [\![p]\!]^T_{\mathbb{N}^k}$. Hence, we conclude.

Termination By construction, we have that $|U| \leq |E|$, and therefore that U is finite. It follows directly that the main loop at lines 9–13 executes at most $|E|$ times. Then, since all the sets manipulated in the algorithm are finite, we can conclude.

7.2.4 Boolean Operators

In order to take care of boolean operations, i.e. disjunction, conjunction and negation, we can simply use standard set operations. First, for negation, we can use set difference. Indeed, a cut satisfies a formula of the form $\neg\varphi$ if and only if it does not satisfy φ. Therefore, the set of cuts satisfying φ is simply given by the set of all cuts from which are removed those cuts that satisfy φ. This can be extended to the tuple representation.

Lemma 7.5

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and a CTL formula φ, we have that:

$$[\![\neg\varphi]\!]^T_{\mathbb{N}^k} = [\![\top]\!]^T_{\mathbb{N}^k} \setminus [\![\varphi]\!]^T_{\mathbb{N}^k}$$

Proof

$$\begin{aligned} [\![\neg\varphi]\!]^T_{\mathbb{N}^k} &= \mathsf{tuple}([\![\neg\varphi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\top]\!]^T_{\mathsf{C}} \setminus [\![\varphi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\top]\!]^T_{\mathsf{C}}) \setminus \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}}) \\ &= [\![\top]\!]^T_{\mathbb{N}^k} \setminus [\![\varphi]\!]^T_{\mathbb{N}^k} \end{aligned}$$

Next, for conjunction, we can use intersection. Indeed, a cut satisfies a formula of the form $\varphi \wedge \psi$ if and only if it satisfies both φ and ψ. The set of cuts satisfying $\varphi \wedge \psi$ is therefore given by the intersection between those cuts that satisfy φ and those that satisfy ψ. Again, this is extended to the tuple representation.

Lemma 7.6

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and two CTL formulae φ and ψ, we have that:

$$[\![\varphi \wedge \psi]\!]^T_{\mathbb{N}^k} = [\![\varphi]\!]^T_{\mathbb{N}^k} \cap [\![\psi]\!]^T_{\mathbb{N}^k}$$

Proof

$$\begin{aligned} [\![\varphi \wedge \psi]\!]^T_{\mathbb{N}^k} &= \mathsf{tuple}([\![\varphi \wedge \psi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}} \cap [\![\psi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}}) \cap \mathsf{tuple}([\![\psi]\!]^T_{\mathsf{C}}) \\ &= [\![\varphi]\!]^T_{\mathbb{N}^k} \cap [\![\psi]\!]^T_{\mathbb{N}^k} \end{aligned}$$

Finally, disjunction can be taken care of with union. Indeed, a cut satisfies $\varphi \vee \psi$ if and only if it satisfies either φ or ψ. The set of cuts satisfying $\varphi \vee \psi$ is therefore given by the union of those cuts that satisfy φ and those cuts that satisfy ψ. Similarly to conjunction and negation, this is also extended to the tuple representation.

Lemma 7.7

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ and two CTL formulae φ and ψ, we have that:

$$[\![\varphi \vee \psi]\!]^T_{\mathbb{N}^k} = [\![\varphi]\!]^T_{\mathbb{N}^k} \cup [\![\psi]\!]^T_{\mathbb{N}^k}$$

Proof

$$\begin{aligned} [\![\varphi \vee \psi]\!]^T_{\mathbb{N}^k} &= \mathsf{tuple}([\![\varphi \vee \psi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}} \cup [\![\psi]\!]^T_{\mathsf{C}}) \\ &= \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}}) \cup \mathsf{tuple}([\![\psi]\!]^T_{\mathsf{C}}) \\ &= [\![\varphi]\!]^T_{\mathbb{N}^k} \cup [\![\psi]\!]^T_{\mathbb{N}^k} \end{aligned}$$

7.2.5 Temporal Modalities

In order to take care of formulae containing temporal modalities, we can use the fixed point characterization of CTL that was presented in Chapter 1. With this characterization, all that is left to do is to devise a way to compute the tuple representation of $\mathsf{pre}(X)$ and $\widetilde{\mathsf{pre}}(X)$. In fact, since $\widetilde{\mathsf{pre}}(X) = Q \setminus \mathsf{pre}(Q \setminus X)$, a symbolic algorithm for $\mathsf{pre}(X)$ is sufficient. For that purpose, we decompose $\mathsf{pre}(X)$ into a function of $\mathsf{pre}_i(X)$, where $\mathsf{pre}_i(X) \stackrel{\mathbf{def}}{=} \{C \in Q \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\}$, i.e. the predecessors of X if only events of process P_i are considered. This decomposition is provided hereafter.

Lemma 7.8

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a subset $X \subseteq Q$, we have that:

$$\mathsf{pre}(X) = \bigcup_{i \in [1,k]} \mathsf{pre}_i(X)$$

Proof

$$\begin{aligned}
\mathsf{pre}(X) &= \{D \in Q \mid \exists C \in X : D \rightarrow C\} \\
&= \{D \in Q \mid \exists C \in X, \exists e \in \mathsf{enabled}(D) : C = D \cup \{e\}\} \\
&\qquad\qquad \text{(by Proposition 4.1)} \\
&= \{D \in Q \mid \exists e \in \mathsf{enabled}(D) : D \cup \{e\} \in X\} \\
&= \{D \in Q \mid \exists e \in \mathsf{enabled}(D) \cap E : D \cup \{e\} \in X\} \\
&= \{D \in Q \mid \exists e \in \mathsf{enabled}(D) \cap \cup_{i \in [1,k]} P_i : D \cup \{e\} \in X\} \\
&= \{D \in Q \mid \exists e \in \textstyle\bigcup_{i \in [1,k]} (\mathsf{enabled}(D) \cap P_i) : D \cup \{e\} \in X\} \\
&= \textstyle\bigcup_{i \in [i,k]} \{D \in Q \mid \exists e \in \mathsf{enabled}(D) \cap P_i : D \cup \{e\} \in X\} \\
&= \textstyle\bigcup_{i \in [i,k]} \mathsf{pre}_i(X)
\end{aligned}$$

The only remaining step is therefore to characterize $\mathsf{pre}_i(X)$. In other words, given $\mathsf{tuple}(X)$, we must compute $\mathsf{tuple}(\mathsf{pre}_i(X))$. This can be done by systematically decrementing the i^{th} component of each tuple $t \in \mathsf{tuple}(X)$, and keeping from the resulting set of tuples, only those that represent actual cuts. Given a tuple of integers $t \in \mathbb{Z}^k$, and a natural number $d \in \mathbb{N}$, we note $t^{[i \gg d]}$ the tuple of integers obtained from t by shifting its i^{th} component of d, i.e. $(t^{[i \gg d]}[i] = t[i] + d) \wedge (\forall j \in [1,k] \setminus \{i\} : t^{[i \gg d]}[j] = t[j])$. Symmetrically, we note $t^{[t \ll d]} \stackrel{\text{def}}{=} t^{[t \gg -d]}$. These operations are extended to sets of tuples as expected. Given a set of tuples of integers $S \subseteq \mathbb{Z}^k$, we note $S^{[i \gg d]} \stackrel{\text{def}}{=} \bigcup_{t \in S} \{t^{[i \gg d]}\}$ and $S^{[i \ll d]} \stackrel{\text{def}}{=} S^{[i \gg -d]}$. Using this notation, we can formalize our intuition as follows.

Lemma 7.9

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a subset $X \subseteq Q$, we have that:

$$\mathsf{tuple}(\mathsf{pre}_i(X)) = \mathsf{tuple}(X)^{[i \ll 1]} \cap \mathsf{tuple}(Q)$$

Proof

$$\begin{aligned}
&\mathsf{tuple}(\mathsf{pre}_i(X)) \\
&\quad = \mathsf{tuple}(\{C \in Q \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\}) \\
&\quad = \mathsf{tuple}(\{C \in Q \mid \exists e \in \mathsf{enabled}(C) \cap P_i : \mathsf{tuple}(C \cup \{e\}) \in \mathsf{tuple}(X)\}) \\
&\quad = \mathsf{tuple}(\{C \in Q \mid \exists e \in \mathsf{enabled}(C) \cap P_i : \mathsf{tuple}(C)^{[i \gg 1]} \in \mathsf{tuple}(X)\}) \\
&\quad = \mathsf{tuple}(\{C \in Q \mid \mathsf{tuple}(C)^{[i \gg 1]} \in \mathsf{tuple}(X)\}) \\
&\quad = \{t \in \mathsf{tuple}(Q) \mid t^{[i \gg 1]} \in \mathsf{tuple}(X)\} \\
&\quad = \{t \in \mathsf{tuple}(Q) \mid t \in \mathsf{tuple}(X)^{[i \ll 1]}\} \\
&\quad = \mathsf{tuple}(X)^{[i \ll 1]} \cap \mathsf{tuple}(Q)
\end{aligned}$$

```
1  function computeTuplesPre(T, S)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes with K_T = ⟨Q, I, L, →⟩ and
            a subset S ⊆ ℕ^k such that S = tuple(X) for some X ⊆ Q
   returns: tuple(pre(X))
2  begin
3      R := ∅
4      forall i ∈ [1, k] do
5          R := R ∪ S^[i≪1]
6      return R ∩ computeTuplesTautology(T)
7  end

8  function computeTuplesPreTilde(T, S)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes with K_T = ⟨Q, I, L, →⟩ and
            a subset S ⊆ ℕ^k such that S = tuple(X) for some X ⊆ Q
   returns: tuple(p̃re(X))
9  begin
10     R := computeTuplesTautology(T)
11     return R \ computeTuplesPre(T, R \ S)
12 end
```

Algorithm 7.4 - Computation of $\mathsf{tuple}(\mathsf{pre}(X))$ and $\mathsf{tuple}(\widetilde{\mathsf{pre}}(X))$

Putting it all back together, we obtain a tuple characterization of $\mathsf{pre}(X)$.

Theorem 7.4 (Tuple Characterization of pre(X))

Given a po-trace $T = \langle E, V_0, \delta, \preceq \rangle$ with $K_T = \langle Q, I, \mathcal{L}, \rightarrow \rangle$, and a subset $X \subseteq Q$, we have that:

$$\mathsf{tuple}(\mathsf{pre}(X)) = \left(\bigcup_{i \in [1,k]} \mathsf{tuple}(X)^{[i \ll 1]} \right) \cap [\![\top]\!]^T_{\mathbb{N}^k}$$

Proof

$$\begin{array}{rcll} \mathsf{tuple}(\mathsf{pre}(X)) & = & \mathsf{tuple}(\bigcup_{i \in [1,k]} \mathsf{pre}_i(X)) & \text{(by Lemma 7.8)} \\ & = & \bigcup_{i \in [1,k]} \mathsf{tuple}(\mathsf{pre}_i(X)) & \\ & = & \bigcup_{i \in [1,k]} (\mathsf{tuple}(X)^{[i \ll 1]} \cap \mathsf{tuple}(Q)) & \text{(by Lemma 7.9)} \\ & = & (\bigcup_{i \in [1,k]} \mathsf{tuple}(X)^{[i \ll 1]}) \cap [\![\top]\!]^T_{\mathbb{N}^k} & \end{array}$$

This yields algorithms for $\mathsf{pre}(X)$ and $\widetilde{\mathsf{pre}}(X)$, as presented in Algorithm 7.4. Note that we tried other ways to $\widetilde{\mathsf{pre}}(X)$. Unfortunately, each attempt lead to a less efficient

algorithm. The algorithms for $\mathsf{EX}, \mathsf{AX}, \mathsf{EU}$ and AU follow, using their fixed point based characterization on $\mathsf{pre}(X)$ and $\widetilde{\mathsf{pre}}(X)$ of Theorem 1.12. Indeed, the lattice of cuts is finite, and both $\mathsf{pre}(X)$ and $\widetilde{\mathsf{pre}}(X)$ are monotonic. However, there is a particular case, where we can do better. Indeed, if the formula is of the form $\mathsf{EF}\,\varphi$, instead of computing $[\![\mathsf{E}(\top\,\mathsf{U}\,\varphi)]\!]^T_{\mathbb{N}^k} = \mathsf{lfp}\ \lambda X \cdot [\![\varphi]\!]^T_{\mathbb{N}^k} \cup ([\![\top]\!]^T_{\mathbb{N}^k} \cap \mathsf{pre}(X))$, we can alternatively compute the downward closure of $[\![\mathsf{EF}\,\varphi]\!]^T_{\mathbb{N}^k}$, which is much more efficient.

Theorem 7.5 (Tuple Characterization of EF)

Given a po-trace $T = \langle E, V_0, \delta, \preceq\rangle$, and a CTL formula φ. We have that:

$$[\![\mathsf{EF}\,\varphi]\!]^T_{\mathbb{N}^k} = {\downarrow}[\![\varphi]\!]^T_{\mathbb{N}^k} \cap [\![\top]\!]^T_{\mathbb{N}^k}$$

Proof

$$\begin{aligned}
[\![\mathsf{EF}\,\varphi]\!]^T_{\mathbb{N}^k} &= \mathsf{tuple}([\![\mathsf{EF}\,\varphi]\!]^T_{\mathsf{C}}) \\
&= \mathsf{tuple}(\{C \in Q \mid \exists D \in Q : (C \rightsquigarrow D) \wedge (\langle T, D\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi)\}) \\
&= \mathsf{tuple}(\{C \in Q \mid \exists D \in Q : (C \rightsquigarrow D) \wedge (D \in [\![\varphi]\!]^T_{\mathsf{C}}\}) \\
&= \mathsf{tuple}(\{C \in Q \mid \exists D \in [\![\varphi]\!]^T_{\mathsf{C}} : C \rightsquigarrow D\}) \\
&= \mathsf{tuple}(\{C \in Q \mid \exists D \in [\![\varphi]\!]^T_{\mathsf{C}} : C \subseteq D\}) \qquad \text{(by Lemma 4.2)} \\
&= \mathsf{tuple}(\{C \in Q \mid \exists D \in [\![\varphi]\!]^T_{\mathsf{C}} : \mathsf{tuple}(C) \leq \mathsf{tuple}(D)\}) \\
&\qquad\qquad \text{(by Proposition 7.1)} \\
&= \{t \in \mathsf{tuple}(Q) \mid \exists t' \in \mathsf{tuple}([\![\varphi]\!]^T_{\mathsf{C}}) : t \leq t'\} \\
&= \{t \in \mathbb{N}^k \mid \exists t' \in [\![\varphi]\!]^T_{\mathbb{N}^k} : t \leq t'\} \cap \mathsf{tuple}(Q) \\
&= {\downarrow}[\![\varphi]\!]^T_{\mathbb{N}^k} \cap [\![\top]\!]^T_{\mathbb{N}^k}
\end{aligned}$$

This can also be used for formulae of the form $\mathsf{AG}\,\varphi$, because of the well known equivalence $[\![\mathsf{AG}\,\varphi]\!]^T_{\mathsf{C}} = [\![\neg\mathsf{EF}\,\neg\varphi]\!]^T_{\mathsf{C}}$, which can be adapted to the tuple representation.

7.2.6 Trace Checking Algorithm

We now give our trace checking algorithm, formalized in Algorithm 7.5. First, the algorithm computes the tuple semantics of φ (line 33). This is done recursively on the structure of φ using the results presented in the previous sections. For $\top$, Algorithm 7.2 is used (line 4). For $\bot$ the algorithm simply returns the empty set (line 6). For propositions, Algorithm 7.3 is used (line 8). For the boolean operators, the results of Section 7.2.4 (lines 9–14) are used. For EX and AX, Algorithm 7.4 is used (lines 15–18). For EF and AG, the algorithm uses Theorem 7.5 (lines 19–22). Finally, for the remaining cases, the algorithm uses the fixed point characterization of CTL provided by Theorem 1.12 (lines 23–28). Once this set is computed, the algorithm then simply checks whether the tuple representation of the initial empty cut belongs to it (line 33).

```
1  function computeTupleRep(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes and a CTL formula φ
   returns: [[φ]]^T_{ℕ^k}
2  begin
3      if φ = ⊤ then
4          R := computeTuplesTautology(T)
5      if φ = ⊥ then
6          R := ∅
7      else if φ = p then
8          R := computeTuplesProposition(T, φ)
9      else if φ = ¬φ_1 then
10         R := computeTuplesTautology(T) \ computeTupleRep(T, φ_1)
11     else if φ = φ_1 ∨ φ_2 then
12         R := computeTupleRep(T, φ_1) ∪ computeTupleRep(T, φ_2)
13     else if φ = φ_1 ∧ φ_2 then
14         R := computeTupleRep(T, φ_1) ∩ computeTupleRep(T, φ_2)
15     else if φ = EX φ_1 then
16         R := computeTuplesPre(T, computeTupleRep(T, φ_1))
17     else if φ = AX φ_1 then
18         R := computeTuplesPreTilde(T, computeTupleRep(T, φ_1))
19     else if φ = EF φ_1 then
20         R :=↓computeTupleRep(T, φ_1) ∩ computeTuplesTautology(T)
21     else if φ = AG φ_1 then
22         R := computeTupleRep(T, ¬EF ¬φ_1)
23     else if φ = E(φ_1 U φ_2) then
24         S_1 := computeTupleRep(T, φ_2), S_2 := computeTupleRep(T, φ_1)
25         R := computeLFP(2^{ℕ^k}, ⊆, λX · S_2 ∪ (S_1 ∩ computeTuplesPre(T, X)))
26     else if φ = A(φ_1 U φ_2) then
27         S_1 := computeTupleRep(T, φ_2), S_2 := computeTupleRep(T, φ_1)
28         R := computeLFP(2^{ℕ^k}, ⊆, λX · S_2 ∪ (S_1 ∩ computeTuplesPreTilde(T, X)))
29     return R
30 end
```

Algorithm 7.5 - CTL trace checking algorithm

```
31 function tupleBasedCTL-TC(T, φ)
   input  : a po-trace T = ⟨E, V_0, δ, ⪯⟩ of k processes and a CTL formula φ
   returns: tt if and only if T ⊨_c φ
32 begin
33 |  return tuple(∅) ∈ computeTupleRep(T, φ)
34 end
```

Algorithm 7.5 - CTL trace checking algorithm (cont'd)

7.3 Symbolic Representation

Many symbolic data structures for representing sets of integers, or natural numbers have been proposed throughout the literature, as e.g. *Natural Decision Diagram* [Boudet and Comon, 1996], *Difference Decision Diagram* [Møller et al., 1999], or *Sharing Trees* [Zampunieris and Le Charlier, 1995]. For our particular application, we propose the use of *Interval Sharing Trees* [Ganty, 2002], an extension of traditional sharing trees. Indeed, an interval sharing tree, or IST for short, symbolically represents unions of multi-rectangles, which arise naturally when representing sets of cuts. For instance, the envelope of a po-trace T of k processes can be viewed as a multi-rectangle in $\mathbb{M}(k)$. Indeed, we have that $\mathsf{env}(T) = [l, u]$ where $l \in \mathbb{N}^k$ is such that $\forall i \in [1, k] : l[i] = 0$ and $u \in \mathbb{N}^k$ is such that $\forall i \in [1, k] : u[i] = |P_i|$. Moreover, the set of tuples appearing after and before an event e, respectively $\mathsf{after}(e)$ and $\mathsf{before}(e)$, can also be viewed as multi-rectangles. For an event $e \in P_i$, we have that $\mathsf{after}(e) = [l', u]$, where $l' \in \mathbb{N}^k$ is such that $l'[i] = |P_i \cap \downarrow e|$ and $\forall j \in [1, k] \setminus \{i\} : l'[j] = 0$, and where u is defined as above. In addition, for an event $e \in P_i$, we have that $\mathsf{before}(e) = [l, u']$, where l is defined as above and where $u' \in \mathbb{N}^k$ is such that $u'[i] = |P_i \cap \downarrow e| - 1$ and $\forall j \in [1, k] \setminus \{i\} : u'[j] = |P_j|$. Another strong point in favor of ISTs, it that all the operations that are required for our trace checking algorithm, i.e. union, intersection, set difference, shifting and downward closure, have been defined on ISTs. For details on those operations, we refer the reader to [Ganty et al., 2006]. For all these reasons, we believe that ISTs are well suited in our context. Note however that other data structures have been studied in a subsequent work by Gabriel Kalyon [Kalyon, 2007].

An IST is basically a directed acyclic graph, where nodes are labelled with an interval of integers. They were introduced in [Ganty, 2002]. Each path in such an IST effectively represents a multi-rectangle. The sharing of common prefixes and suffixes of multi-rectangles allows to obtain a compact representation. Note that the counter-part

of this compactness is that most of the operations cannot be computed in polynomial time in general. Hence, (most of) the symbolic algorithms to manipulate ISTs are exponential in their worst case. We refer the reader to [Ganty et al., 2006] for a detailed complexity analysis. However, those algorithms are in general far from their worst case and in practice, ISTs have been shown to be more efficient than other known data-structures (to represent subsets of $\mathbb{N}^k$) both in execution time and memory, in the context of model checking [Ammirati et al., 2002]. The formal definition of IST follows.

Definition 7.4 (Interval Sharing Tree)

An interval sharing tree (IST) of k layers is a tuple $S = \langle N, \iota, \alpha \rangle$ where:

- $N = N_0 \cup N_1 \cup N_2 \cup ... \cup N_k \cup N_{k+1}$ is the finite set of nodes, partitioned into $k+2$ disjoint subsets N_i called layers with $N_0 \stackrel{\text{def}}{=} \{n_\top\}$ and $N_{k+1} \stackrel{\text{def}}{=} \{n_\bot\}$
- $\iota \in N \mapsto \mathbb{I} \cup \{\top, \bot\}$ is the labelling function such that $\iota(n) = \top$, respectively $\bot$, if and only if $n = n_\top$, respectively $n_\bot$.
- $\alpha \in N \mapsto 2^N$ is the successor function such that:
  - (i) $\alpha(n_\bot) = \varnothing$
  - (ii) $\forall i \in [0,k], \forall n \in N_i : \alpha(n) \subseteq N_{i+1} \wedge \alpha(n) \neq \varnothing$
  - (iii) $\forall n \in N, \forall n', n'' \in \alpha(n) : (n' \neq n'') \Rightarrow (\iota(n') \neq \iota(n''))$
  - (iv) $\forall i \in [0,k), \forall n, n' \in N_i : (n \neq n' \wedge \iota(n) = \iota(n')) \Rightarrow (\alpha(n) \neq \alpha(n'))$

In other words, an IST is a directed acyclic graph where the nodes are labelled with intervals of integers except for two special node $n_\top$ and $n_\bot$, such that (i) the only node $n_\bot$ of the last layer has no successors, (ii) all nodes from layer $i \leq k$ have their successors in layer $i+1$, (iii) a node cannot have two successors with the same label, (iv) two nodes with the same label in the same layer do not have the same successors. A *path* of an IST is a sequence $n_0 n_1 \ldots n_{k+1}$ of nodes such that $n_0 = n_\top$, $n_{k+1} = n_\bot$ and such that $\forall i \in [0,k] : n_{i+1} \in \alpha(n_i)$. We note $\mathsf{path}(S)$ the set of such paths. A tuple $t \in \mathbb{Z}^k$ is accepted by an IST S if and only if there exists a path $n_0 n_1 \ldots n_{k+1} \in \mathsf{path}(S)$ such that $\forall i \in [1,k] : t[i] \in \iota(n_i)$.

Example 7.7

Figure 7.7 illustrates the use of ISTs for computing the set of tuples representing all the cuts of the po-trace from Figure 4.4.

Figure 7.7 - Computation of the IST for $[\![\top]\!]^T_{\mathbb{N}^k}$ on the po-trace of Figure 4.4

Experiment				IST	NuSMV
Model	Proc.	Events	Property	Time	Time
Peterson	2	2000	φ_1	0.46s	349.57s
	2	5000	φ_1	7.53s	↑↑
	2	15000	φ_1	189.65s	↑↑
PetersonN	2	2000	φ_2	0.20s	294.46s
	2	5000	φ_2	6.44s	↑↑
	2	20000	φ_2	390.90s	↑↑
	5	1000	φ_2	2.04s	13.74s
	5	1500	φ_2	6.82s	↑↑
	5	5000	φ_2	176.62s	↑↑
	10	1500	φ_2	7.53s	150.23s
	10	2000	φ_2	27.01s	↑↑
	10	5000	φ_2	147.89s	↑↑
ABProtocol	2	1000	φ_3	13.60s	297.28s
	2	2000	φ_3	27.56s	↑↑
	2	5000	φ_3	257.29s	↑↑
Philosopher	3	100	φ_4	0.15s	6.36
	3	200	φ_4	1.11s	↑↑
	3	2000	φ_4	366.22s	↑↑
	5	100	φ_4	0.25s	↑↑
	5	500	φ_4	125.56s	↑↑
	10	100	φ_4	1.67s	↑↑
	10	200	φ_4	26.94s	↑↑

Figure 7.8 - Experimental results (↑↑ indicates $>$ 10 min.)

7.4 Experimental results

In this section, we experimentally validate our method. We compare our symbolic approach, using ISTs with a state-of-the-art symbolic model checking of traces using the tool NuSMV [Cimatti et al., 1999; Cimatti et al., 2002]. It is important to note that, through these experiments, we did not try to compare the use of ISTs in particular, with other data structures, like BDDs, but rather to validate our technique for computing the set of tuples corresponding to a given CTL formula φ. In fact, as shown in [Kalyon, 2007], BDDs can also be used for that, using a binary encoding of the tuples.

We considered several classical academic examples of distributed systems and compared the running time of our early prototype against NuSMV. Running time was limited to 10 minutes. This seems to be a reasonable assumption considering that the testing should be achieved on a large number of traces. Those results are presented in Figure 7.8. On all the examples we considered, memory consumption was not an issue. The ISTs manipulated in these examples contain no more than 7000 nodes. It is therefore not mentioned.

The first example we considered was the *Peterson* mutual exclusion protocol with two processes, where communication is done through shared variables. On this example, we tested that mutual exclusion was satisfied. This was done using the CTL formula $\varphi_1 \stackrel{\text{def}}{=} \neg\mathsf{EF}(\mathsf{crit}_1 \wedge \mathsf{crit}_2)$. Even on this relatively small example, we can already see a big difference in running time: NuSMV runs out of time after 2000 events, whereas out tool can handle 15000 events in the allotted time. We also considered a generalization of this protocol for n processes (PetersonN), implementing the same mutual exclusion property. This was done using the CTL formula $\varphi_2 \stackrel{\text{def}}{=} \neg\mathsf{EF}(\bigwedge_{i\in[1,n]} \mathsf{crit}_i)$. We experimented on 2, 5 and 10 processes. Again, we can see that our approach using ISTs outperforms the traditional CTL symbolic model checking using BDDs.

The third model we considered was the *alternating-bit protocol* between two processes, i.e. a sender and a receiver. This time the communication is achieved using asynchronous channels. We verified that every message tagged with a 0 is followed by one with the same tag. This was done using the CTL formula $\varphi_3 \stackrel{\text{def}}{=} \mathsf{AG}(\mathsf{sent}_0 \Rightarrow \mathsf{F}(\mathsf{recv}_0 \vee \mathsf{eot}))$. This formula is a bit more complicated. Nonetheless, our method is still scalable up to 5000 events, whereas NuSMV stops after 1000.

The last example we considered was the *Dining Philosophers* problem. We considered 3, 5 and 10 philosophers. We verified that whenever philosopher 1 is eating, either he keeps eating until the end of the trace or his left neighbour cannot eat until he stops. This is done using the CTL formula $\varphi_4 \stackrel{\text{def}}{=} \mathsf{AG}(\mathsf{eat}_1 \Rightarrow (\mathsf{AG}(\mathsf{eat}_1) \vee \mathsf{A}(\neg\mathsf{eat}_0\ \mathsf{U}\ \neg\mathsf{eat}_1))$. We deliberately chose a complex formula to test the robustness of our approach. On this example, NuSMV can only handle 3 philosophers with 100 events, with the (too complex) property in the allotted time whereas we can still manage to terminate the analysis on some instances of respectable size. This can be explained by the fact that, in this model, the processes are more independant, thus leading to more interleavings.

For each example, we have computed the size of the lattice of cuts. In the 10 minutes of allotted times, our prototype is capable of handling instances of up to 10^{10} cuts, whereas NuSMV stops at 10^5. This leads us to conclude that our approach is more scalable for this problem.

Chapter

8

Case Study: A Canal Lock Controller

« *In theory, there is no difference between theory and practice. But, in practice, there is.* »

Lawrence Peter "Yogi" Berra

In Chapter 2, we presented dSL as a means to ease the development of distributed reactive control systems. Then, in Chapter 3, we explained how model checking could be applied, by defining a formal semantics for dSL programs. Moreover, in Chapter 4, we explained how dSL programs could be instrumented in order to collect distributed execution traces, modeled as *partial order traces*. Finally, in the subsequent chapters, we studied how such partial order traces, could be analysed efficiently using classical formal logic. In this chapter, we illustrate the concepts developed throughout this dissertation on a concrete case study: the design and validation of a canal locks controller.

This chapter is structured as follows. First, in Section 8.1, we describe the problem statement, i.e. system that we are trying to control, as well as the property this system has to meet in order to be considered correct. We follow, in Section 8.2, with the design of the controller using dSL. Then, in Section 8.3, we examine its model checking using the techniques presented in Chapter 3. Finally, in Section 8.4, we turn our attention to the testing of this controller using the techniques presented in Chapters 4 to 7.

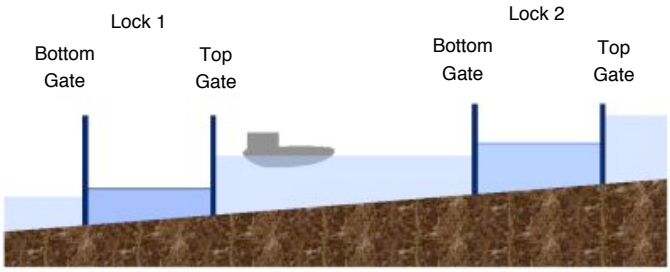


Figure 8.1 - A canal lock system

8.1 Problem Statement

The system is composed of two consecutive canal locks. As illustrated in figure 8.1, each lock is composed of two gates, a top and a bottom one. In between the top and the bottom gates of each lock, the water level can be controlled (i.e. the inside of a lock can be filled or emptied). The different commands of this system (opening/closing a gate, emptying/filling a lock) can be accessed via a control panel.

For this system to function properly, several constraints must be satisfied. First of all, two consecutive gates cannot be opened at the same time. Then, of course, a gate can only be opened if the water level on each side is the same. Finally, the water level inside a lock can only be changed if both of its gates are closed. The purpose of the controller is to ensure that these constraints are verified at all time. Whenever a command is introduced via the control panel, before taking the appropriate action, the controller must first check that it will not jeopardize the safety of the system, in which case, the action is not taken, and a red light indicator on the control panel is switched on to indicate an invalid command.

8.2 Designing the Controller

The main idea to implement the controller in dSL is the following. We start by modeling the variables of the system using classes. As illustrated in Figure 8.2(a), there are two main classes: the class `GATE` and the class `LOCK`. Each command `xxx` of the system (opening/closing a gate, emptying/filling a lock) is included in the appropriate class as two variables. The first (`xxxCommand`) activates the motor/valve and the second (`xxxDirection`) determines the direction (up/down or fill/empty). On top of that, a special variables `xxxOrderGiven` is added. Whenever an order is given, the correspond-

```
 2 CLASS GATE
 3   motorCommand, motorDirection        : BOOL;
 4   opened, closed, motorOrderGiven     : BOOL;
 5   buttonOpen, buttonClose             : BOOL;
 6 END_CLASS
 7
 8 CLASS LOCK
 9   valveCommand, valveDirection        : BOOL;
10   levelUp, levelDown, valveOrderGiven : BOOL;
11   bottomGate, topGate                 : GATE;
12   buttonFill, buttonEmpty             : BOOL;
13 END_CLASS
```

(a) Classes

```
127 WHEN lock2.bottomGate.buttonOpen THEN
128   IF (~lock1.topGate.closed) AND (NOT lock1.topGate.motorOrderGiven) AND
129      (~lock2.topGate.closed) AND (NOT lock2.topGate.motorOrderGiven) AND
130      (~lock2.levelDown) AND (NOT lock2.valveOrderGiven)
131   THEN
132     notAllowed := FALSE;
133     lock2.bottomGate.motorOrderGiven := TRUE;
134     LAUNCH lock2.bottomGate<-move(TRUE);
135   ELSE
136     notAllowed := TRUE;
137   END_IF;
138 END_WHEN
```

(b) Event dealing with the opening of the bottom gate of the upper lock

Figure 8.2 - Excerpt from the dSL source code of the canal lock controller

ing boolean variable `xxxOrderGiven` is set. Then, when a command is received from the control panel, the controller simply checks that all the requirements are satisfied and, using those extra variables, that no command on the checked gates and water valves is under way. The `xxxOrderGiven` variables are, of course, reset when an order is completed. In this implementation, commands are executed using events. As an example, Figure 8.2(c) presents the event dealing with the opening of the bottom gate of the upper lock. The distribution is composed of three sites : `lowerLock`, `upperLock` and `controlPanel`. The first two sites correspond to the two locks. As illustrated in Figure 8.2(c), this is where all actuators and sensors are localised. As illustrated in Figure 8.2(d), the remaining execution site corresponds to the control panel, where all

```
22 SITE lowerLock
23   INPUT lock1.bottomGate.opened          : 1.0.1;
24   INPUT lock1.bottomGate.closed          : 1.0.2;
25   INPUT lock1.topGate.opened             : 1.0.3;
26   INPUT lock1.topGate.closed             : 1.0.4;
27   INPUT lock1.levelDown                  : 1.0.5;
28   INPUT lock1.levelUp                    : 1.0.6;
29   OUTPUT lock1.bottomGate.motorCommand   : 2.0.1;
30   OUTPUT lock1.bottomGate.motorDirection : 2.0.2;
31   OUTPUT lock1.topGate.motorCommand      : 2.0.3;
32   OUTPUT lock1.topGate.motorDirection    : 2.0.4;
33   OUTPUT lock1.valveCommand              : 2.0.5;
34   OUTPUT lock1.valveDirection            : 2.0.6;
35 END_SITE
```

(c) The `lowerLock` site

```
52 SITE controlPanel
53   INPUT lock1.bottomGate.buttonOpen      : 1.0.1;
54   INPUT lock1.bottomGate.buttonClose     : 1.0.2;
55   INPUT lock1.topGate.buttonOpen         : 1.0.3;
56   INPUT lock1.topGate.buttonClose        : 1.0.4;
57   INPUT lock1.buttonFill                 : 1.0.5;
58   INPUT lock1.buttonEmpty                : 1.0.6;
59   INPUT lock2.bottomGate.buttonOpen      : 1.0.7;
60   INPUT lock2.bottomGate.buttonClose     : 1.0.8;
61   INPUT lock2.topGate.buttonOpen         : 1.0.9;
62   INPUT lock2.topGate.buttonClose        : 1.0.10;
63   INPUT lock2.buttonFill                 : 1.0.11;
64   INPUT lock2.buttonEmpty                : 1.0.12;
65   OUTPUT notAllowed                      : 2.0.1;
66 END_SITE
```

(d) The `controlPanel` site

Figure 8.2 - Excerpt from the dSL source code of the canal locks controller (cont'd)

```
1 inline flip_flop(flipped, flopped, command, direction) {
2     if :: command && flipped ->
3           if :: direction == FLIP -> skip;
4              :: direction == FLOP -> flipped = false;
5           fi;
6        :: command && !flipped && !flopped ->
7           if :: direction == FLIP -> flipped = true;
8              :: direction == FLOP -> flopped = true;
9              :: skip;
10          fi;
11       :: command && flopped ->
12          if :: direction == FLIP -> flipped = false;
13             :: direction == FLOP -> skip;
14          fi;
15    fi;
16 }
```

Figure 8.2 - Flip-Flop behaviour

command buttons and led are localized. The complete dSL source code of this example can be found in Appendix D.

Note that for all the `xxxOrderGiven` variables, the $\sim$ operator cannot be used. For example, in Figure 8.2(b), if the variable `lock2.topGate.motorOrderGiven` were tilded, when an order is given to open the bottom gate of the upper lock, the controller would check if variable $\sim$`lock2.topGate.motorOrderGiven` is false. However, even in this case, the actual variable might be true but, because of communication delays, the `upperLock` site might not know it yet. The controller would then allow the bottom gate of upper lock to open while the top gate is either opening or about to open.

8.3 Model Checking the Controller

The first step towards model checking the controller presented in the previous section is to translate it to Promela. This is done using the formal semantics developed in Chapter 3. Then, it is necessary to add the behaviour of the environment to this Promela model. For that, we need to take into account several things: (*i*) the gates, (*ii*) the water level inside the locks and finally (*iii*), the operator acting on the control panel. The first two are modelled using a classical non deterministic *flip-flop* behaviour. This behaviour has three states: *flipped*, *flopped* and *changing*. In the case of the gates, *flipped*, respectively *flopped*, corresponds to the variable `opened`, respectively `closed`, being true. On the other hand, for the water levels in the locks, *flipped*, respectively

Sites	Button	Channels	Property	Verified	Time	States	Memory
1	2	Instant	φ_1	$\surd$	3.25s	1.41e+04	1.432 MB
2	2	Instant	φ_1	$\times$	14.52s	8.37e+04	3.064 MB
3	2	Instant	φ_1	$\times$	8.00s	9.54e+04	3.569 MB
7	2	Instant	φ_1	$\times$	9.74s	7.82e+04	3.645 MB
11	2	Instant	φ_1	$\times$	5m 38.48s	1.41e+06	68.557 MB
1	1	Normal	φ_2	$\surd$	7.80s	3.98e+04	2.457 MB
2	1	Normal	φ_2	$\surd$	43.22s	1.29e+05	4.908 MB
3	1	Normal	φ_2	$\times$	37.13s	1.39e+05	5.311 MB
7	1	Normal	φ_2	$\times$	21m 14.36s	1.91e+06	77.581 MB
11	1	Normal	φ_2	$\times$	54.93s	2.07e+05	10.383 MB

Figure 8.3 - Results of the verification of the canal lock controller

flopped, corresponds to the variable `levelDown`, respectively `levelUp`, being true. The Promela model for this behaviour is presented in Figure 8.2. The control panel operator is then modeled as follows. It repeatedly choses one of the twelve buttons of the control panel non-deterministically and switches it, i.e. releases it if it was pressed or presses it if it was released.

The second step is to model the safety requirements imposed on the system in LTL. This is expressed using the following formula:

$$\varphi_1 \stackrel{\text{def}}{=} \mathsf{G}\,\neg \left(\begin{array}{c} (\neg \texttt{lock1.bottomGate.closed} \wedge \neg \texttt{lock1.topGate.closed}) \vee \\ (\neg \texttt{lock1.topGate.closed} \wedge \neg \texttt{lock2.bottomGate.closed}) \vee \\ (\neg \texttt{lock2.bottomGate.closed} \wedge \neg \texttt{lock2.topGate.closed}) \vee \\ (\neg \texttt{lock1.bottomGate.closed} \wedge \neg \texttt{lock1.levelDown}) \vee \\ (\neg \texttt{lock1.topGate.closed} \wedge \neg \texttt{lock1.levelUp}) \vee \\ (\neg \texttt{lock2.bottomGate.closed} \wedge \neg \texttt{lock2.levelDown}) \vee \\ (\neg \texttt{lock2.topGate.closed} \wedge \neg \texttt{lock2.levelUp}) \end{array} \right)$$

For the purpose of our experiments, we also introduced a restricted version that only expresses the fact that the two middle gates (i.e the top gate of the first lock and the bottom gate of the second) are not opened at the same time. This is expressed in LTL using the following formula:

$$\varphi_2 \stackrel{\text{def}}{=} \mathsf{G}\,\neg(\neg \texttt{lock1.topGate.closed} \wedge \neg \texttt{lock2.bottomGate.closed})$$

Finally, the model was verified using the tool Spin. On top of the original localization table (Λ) presented in Appendix D containing 3 execution sites, several other localization tables were considered, with respectively 1, 2, 7 and 11 execution sites.

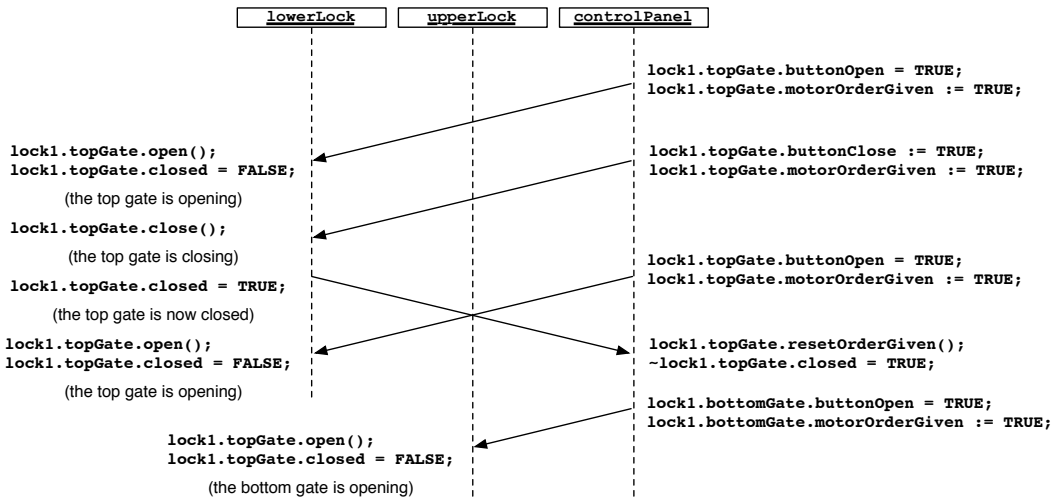


Figure 8.4 - Error in the canal locks controller

Figure 8.3 shows some representative results corresponding to the verification of the controller, using Spin version 4.2.4 with partial order reduction, on a 3 GHz Intel Xeon machine with 2GB of memory. The first column indicates the number of execution sites in the localization table. The second column indicates the maximum number of times each button can be pressed. The next column indicates whether or not messages are taken from their queues as soon as possible. *Instant* indicates channels of size 0, (equivalent to synchronous *rendez-vous* communication in Promela), and *Normal* indicates channels of size 1. The fourth column indicates which property was checked. The four remaining columns show respectively, whether the property was verified or not, the time needed for the verification, the number of states explored and the memory used. Surprisingly, as exhibited in the results of Figure 8.3, the controller presented in Appendix D is faulty. Indeed, as shown by the error trace depicted in Figure 8.4, two consecutive gates can be opened at the same time. In this case, three orders are given to the top gate of the first lock: an order to open, immediately followed by an order to close (before the gate is completely opened) and finally an order to open again. Because of communications, the `resetOrderGiven()` and the value of `lock1.topGateClosed` are delayed (respectively because of the `LAUNCH` and '~'). Therefore when the order to open the bottom gate of the second lock is given, the controller believes that the top gate of the first lock is closed and that no order has been given to it. This allows the opening of the bottom gate of the second lock, which violates the constraints. An easy way to correct this, would be to allow a command to a gate (or a water level) only if its `xxxOrderGiven` is false (in other words, only allowing one order at a time). However,

```
1 WHEN lock2.bottomGate.buttonOpen AND NOT disabled THEN
2     disabled := TRUE;
3     LAUNCH open_bottom_gate_lock2;
4 END_WHEN
5
6 SEQUENCE open_bottom_gate_lock2()
7 VAR
8     check : BOOL;
9 END_VAR
10    check := (lock2.topGate.closed AND lock2.levelDown);
11    check := (check AND lock1.topGate.closed);
12    IF check THEN
13        notAllowed := FALSE;
14        LAUNCH lock2.bottomGate<-open();
15    ELSE
16        notAllowed := TRUE;
17    END_IF;
18    disabled := FALSE;
19 END_SEQUENCE
```

Figure 8.5 - dSL code dealing with the opening of the bottom gate of the upper lock

this would not be a viable solution. Indeed, imagine a boat breaks down while the gate is closing, the controller would not allow to open a gate until it is completely closed, and the boat would be crushed! So, instead of blocking all commands while an order is processed, it is far better to disable the commands only during the time needed to verify the (distributed) constraints using an additional boolean variable `disabled`. To achieve this, a sequential execution checks that the issued command can be executed, by migrating the condition to all intervening sites. As illustrated in figure 8.5, this is done by means of a sequence that first evaluates, in a local variable `check`, that all the conditions are satisfied. In the example of Figure 8.5, the first part of the constraint (`check := (lock2.topGate.closed and lock2.levelDown);`) will be evaluated on the site `upperLock`, then the value of `check` will be migrated to the site `lowerLock` to evaluate the second part (`check := (check and lock1.topGate.closed);`). Since the control panel is disabled during this task, we can be sure that the variable `check` is true if and only if the constraints are satisfied, in which case, the corresponding action(s) is (are) taken.

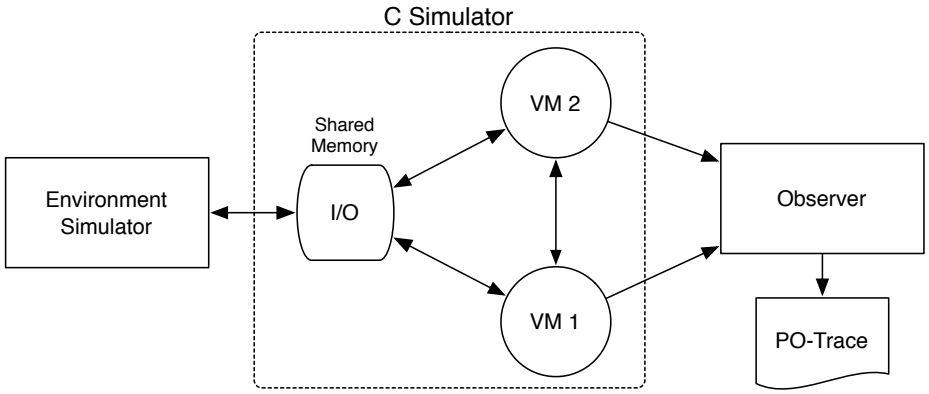


Figure 8.6 - Overview of the C simulator

8.4 Testing the Controller

The first step towards testing the canal lock controller is to collect po-traces from that controller. For that purpose, similarly to what is done in model checking, we need to close the system, i.e. provide an environment in which to run the controller. The real physical environment could of course be used. However, early in the project, this physical environment may not exist yet, in which case, it must be replaced by a realistic simulation of its behaviour. With this in mind, we developed a back-end to the dSL compiler that, given a dSL program, generates a multi-process C program simulating its behaviour. This program is automatically instrumented using the vector clocks algorithm of Chapter 4 so that, when executed, the events of interest are sent to the observer where they are collected into a po-trace. As illustrated in Figure 8.6, for each execution site of the program, a dedicated C process is generated using the formal semantics presented in Chapter 3. The network communications are simulated using UNIX pipes, and the environment is modeled using a shared memory. In this fashion, the environment can be simulated using any home-made C program(s). Note that, as an added bonus, this simulator can also be used as a temporary development platform in the initial steps of the project.

Similarly to what was done for the model checking, the gates and water level inside the locks were modeled as a *flip-flop* behaviour, and the control panel operator was modeled as a program repeatedly pressing or releasing one of the twelve buttons chosen at random. Using this technique, we collected 10 po-traces, by executing the system for approximately 30 seconds (corresponding to approximately 3700 events). We verified

Experiment			LTL	CTL
Proc.	Events	Error	Time	Time
3	3709	YES	3.62 sec.	40.15 sec.
3	3797	NO	5.14 sec.	39.29 sec.
3	3715	YES	3.44 sec.	37.49 sec.
3	3959	NO	6.88 sec.	47.58 sec.
3	3876	NO	5.58 sec.	42.57 sec.
3	3555	NO	4.74 sec.	31.02 sec.
3	3716	YES	3.54 sec.	33.56 sec.
3	3797	NO	5.92 sec.	40.72 sec.
3	3795	YES	3.47 sec.	41.40 sec.
3	3684	NO	6.01 sec.	41.81 sec.

Figure 8.7 - Results of the testing of the canal lock controller

the global property φ_1 using the LTL symbolic trace checking algorithm[1] of Chapter 6, and a CTL equivalent using the symbolic CTL trace checking algorithm of Chapter 7. As explained in Section 8.2, the controller was distributed on three execution sites, i.e. one for each lock and one for the control panel. The trace analysis was done on a 2.16GHz Intel Core 2 Duo machine with 2GB of memory. The results of these experiments are presented in Figure 8.7. The first 3 columns describe the po-traces: the number of processes, the number of events, and whether or not a violation of global property was detected. The remaining two columns show the time needed for the trace checking. Out of the 10 po-traces, 4 exhibited a violation of the property. After careful review of those po-traces, we determined that, in each case, the error followed the same pattern as the error presented in Figure 8.4, i.e. using a quick succession of contradictory commands, the controller can be fooled into executing a command that jeopardizes the safety of the system. For instance, in two of the faulty po-traces, the water level of the upper lock was permitted to change even though the bottom gate was not completely closed.

As can be observed in this figure, on these experiments, the symbolic trace checking algorithm for LTL is faster than its CTL counterpart by approximately a factor of ten. We believe that this is due to the particular nature of the po-traces. Indeed, in a dSL system, each time a tilded variable is assigned in one process, a message is broadcast to the other processes. In the canal locks controller, this happens at each cycle, since most of the sensors in the system are used tilded. Therefore, this yields po-traces with a lot of communication edges. Unfortunately, in the symbolic CTL algorithm, when

[1] We also tried to apply the explicit LTL algorithm, i.e. Algorithm 6.2, but in each instance, it did not terminate in under 10 minutes

Experiment		LTL	CTL
Proc.	Prefix	Time	Time
3	500	0.126 sec.	0.077 sec.
3	1000	0.414 sec.	0.499 sec.
3	1500	0.877 sec.	1.791 sec.
3	2000	1.453 sec.	5.011 sec.
3	2500	2.198 sec.	11.270 sec.
3	3000	3.120 sec.	20.601 sec.
3	3500	4.118 sec.	35.868 sec.

Figure 8.8 - Detailed analysis of one trace

building $[\![\top]\!]^T_{\mathsf{C}}$, each of those communication edges must be taken into account and this slows down the construction of $[\![\top]\!]^T_{\mathsf{C}}$ considerably. As a matter of fact, during the experiments, almost all the time needed for the trace checking is spent in the construction of $[\![\top]\!]^T_{\mathsf{C}}$. On the other hand, those communication edges significantly constraint the partial order on the events of the trace, therefore reducing the total number of cuts. That is why the symbolic LTL algorithm performs so well in this case.

We also examined how the LTL and CTL compare with respect to the number of events. For that purpose, we picked one of trace and examined the time needed to analyse prefixes of this trace. The result for this experiment are presented in Figure 8.8. As can be observed in this figure, the two algorithms have similar performances when dealing with short prefixes (500 to 1500 events). However, when the size of the prefix grows, the advantage of LTL over CTL becomes more evident.

This seems to indicate that for dSL programs, whenever possible (e.g. for safety properties), the designer should use LTL instead of CTL as the specification language.

Conclusion

« If we knew what it was we were doing,
it would not be called research, would it? »

Albert Enstein

As argued in the introduction, *distributed reactive control systems* are intrinsically difficult to design. When dealing with critical applications, failures in those control systems can have disastrous consequences. This motivated our initial goal of providing the designer with the necessary tools to ease their *development*, and *validation*.

We started our work by tackling the *development* of those systems. For that purpose, we introduced dSL, a full-featured programming language adapted from the industrial world, providing *transparent code distribution*. As we have seen, this mechanism considerably eases the development of distributed reactive control systems, by taking care of the *communication* aspects, thus allowing the designer to concentrate on the more important *functional* aspect of the system. This was a first step. We then turned our attention to the *validation* of systems developed in this paradigm.

The first and most promising validation techniques we considered was *model checking*, i.e. an exhaustive and automatic verification that the system fulfills the required properties. For that purpose, we presented, and studied, a formal semantics for dSL programs. This allowed us to formalize the model checking problem for dSL programs. We saw that, when dealing with LTL-X specifications, the property could be checked for all distributions in one step by only examining the *maximal* distribution. However, as we have seen, in general, the model checking problem is undecidable, both for LTL and CTL. We therefore introduced a bounded semantics. This semantics, in turn, allowed us to define a translation of dSL to Promela, the input language of the model checker Spin, which makes the model checking of dSL programs possible in practice. We then turned our attention to a less ambitious but more practical approach: *testing*.

We started this endeavour, by defining a formal model to capture distributed execution traces, namely the model of *partial order traces*, central in our work. We studied in great detail the problem of determining if such a partial order trace satisfies a given property. In particular, we examined non-temporal properties, in which case, this problem is called the *predicate detection problem*, and temporal properties, in which case this problem is called the *trace checking problem*.

For the non-temporal case, we saw that the problem is NP-complete in the general case. However, we also showed that if the formula is in disjunctive normal form, the problem becomes polynomial, which is fortunate. Indeed, in the context of testing, a formula is generally used to test a large number of po-traces. Therefore, the time taken to transform a formula in disjunctive normal form will, in general, be compensated by the time gained during the predicate detection.

For the trace checking problem, we examined the two classical temporal logic LTL and CTL. In the case of LTL, we showed that the trace checking problem was coNP-complete. However, inspired by classical model checking algorithm, we were able to define a symbolic trace checking algorithm, efficient in practice. For that purpose, we showed that we had to transform the monitor of the formula into a *determined* one, which can be a costly operation. However, similarly to the predicate detection problem, this only has to be done once. The determined monitor then can be used to test as many po-traces as necessary.

Finally, we turned to CTL, and showed that, in this case, the trace checking problem was even more difficult, i.e. PSPACE-complete. But again, inspired by the classical model checking algorithms, we devised a symbolic approach based on tuples representation of sets of cuts. This lead us to the definition of a trace checking algorithm, which works quite well in practice. Unfortunately, as shown by the experimental results of Chapter 8, this algorithm seems to be outperformed by the LTL trace checking algorithm, in the case of dSL systems. We therefore concluded that, whenever possible, LTL should be privileged.

Personal Contribution

Since most of the work included in this dissertation is collaborative, we feel the need to properly delimit the boundaries of our personal contribution. In Chapter 2, the definition of the formal semantics, and the results on LTL-X is a joint effort with Bram De Wachter, Alexandre Genon and Thierry Massart, and are also presented in Bram's thesis. The technical details of the one-split simulation lemma were handled by Alexan-

dre Genon, for his DEA thesis [Genon, 2004]. The undecidability result, as well as the definition of the bounded semantics are my work. The definition of the translation to Promela and its implementation in the compiler, however, should be credited to Bram De Wachter.

In Chapter 4, the definition of the model of po-trace and the complexity results on CTL and LTL trace checking are the fruit of a joint effort with Laurent Van Begin and Thierry Massart. In Chapter 5, the syntactic characterization of predicates, as well as the result on DNF predicates, are my work. In Chapter 6, the construction of determined monitor is also the result of my work. The definition of the symbolic composition was based on an original idea by Alexandre Genon, later developed collaboratively. Finally, the symbolic CTL trace checking algorithm in Chapter 7 are, for the most parts, my work. The implementation of the prototype used to obtain the experimental results of Chapters 6 to 8 is also my work.

Future works

In Chapter 2, guided by the simulation results obtained in Section 3.2, we concentrated on the verification of LTL properties through a translation to Promela. Of course, one could also investigate the verification of CTL properties. In fact, we recently started to investigate this problem using the model checker XTL [Leuschel and Massart, 2000]. In this approach, the formal operational semantics rule are directly described in Prolog [Bramer, 2005]. A dSL program is then translated in a Prolog model. The XTL model checker can then be used to verify CTL properties on this model. One could also consider other approaches, based for instance on a translation to SMV [McMillan, 1993]

In Chapter 4, we defined the model of po-traces, and explained how to specify properties using PBL, LTL and CTL formulae. We also saw that the detection of PBL predicates could easily be expressed in LTL or CTL. A natural extension of this is to examine the expressive power of LTL and CTL compared to each other. For total order traces, as we have seen in the proof of Theorem 4.7, LTL and CTL are equally expressive. On Kripke structures however, it is well known that LTL and CTL are disjoint [Clarke et al., 1999], i.e. there exists CTL formulae that cannot be expressed in LTL and vice versa. However, in the case of po-traces, this remains an open question. We investigated the straightforward translation used in Theorem 4.7, but it does not work (the difficulty lies in the X operator).

Another interesting line of research would be to try and adapt the ideas behind

the symbolic composition, presented in Chapter 6 to perform full-blown LTL model checking (i.e. on arbitrary Kripke structure). The main difficulty, however, is to handle loops. Indeed, the symbolic composition we defined considerably relies on the partial ordering that inherently exists between the events of a po-trace. One possible solution to overcome this problem would be to use unfoldings [McMillan, 1993; Esparza and Heljanko, 2001].

Finally, we could also investigate how, given a description of the system to be controlled, one could automatically synthesize a distributed control strategy that ensures that a given specification is de facto satisfied. This control strategy could then be used as a basis for implementing a dSL controller. However, for this to become a reality, much work remains to be done...

Appendix

A

Grammars of dSL

In the following grammars, terminals are noted in upper case, using `true type` font. Non-terminal nodes are noted in lower case, between angular brackets $\langle\cdot\rangle$. We also suppose that $\langle$id$\rangle$ and $\langle$nb$\rangle$, respectively denoting identifiers and numbers, are defined in the lexical analyser.

A.1 Full Grammar

$\langle$dsl_program$\rangle$ ::= $\langle$decl_list$\rangle$

$\langle$decl_list$\rangle$ ::= $\langle$decl$\rangle$ $\langle$decl_list$\rangle$

| ε

$\langle$decl$\rangle$ ::= $\langle$class_decl$\rangle$

| $\langle$var_decl$\rangle$

| $\langle$site_decl$\rangle$

| $\langle$method_decl$\rangle$

| $\langle$event_decl$\rangle$

| $\langle$sequence_decl$\rangle$

$\langle$class_decl$\rangle$ ::= `CLASS` $\langle$id$\rangle$
$\langle$var_list$\rangle$
`END_CLASS`

$\langle$var_decl$\rangle$ ::= `VAR`
$\langle$var_list$\rangle$
`END_VAR`

⟨site_decl⟩ ::= `SITE` ⟨id⟩
⟨localization_list⟩
`END_SITE`

⟨method_decl⟩ ::= `METHOD` ⟨id⟩ :: ⟨id⟩ `(` ⟨param_decl⟩ `)`
⟨var_decl⟩
⟨block⟩
`END_METHOD`

⟨sequence_decl⟩ ::= `SEQUENCE` ⟨id⟩ `(` ⟨param_decl⟩ `)`
⟨var_decl⟩
⟨block⟩
`END_SEQUENCE`

⟨event_decl⟩ ::= `WHEN` ⟨rhside⟩ `THEN`
⟨block⟩
`END_WHEN`

| `WHEN IN` ⟨id⟩ ⟨rhside⟩ `THEN`
⟨block⟩
`END_WHEN`

⟨var_list⟩ ::= ⟨id_list⟩ : ⟨type⟩ ; ⟨var_list⟩

| ε

⟨localization_list⟩ ::= ⟨localization⟩ `;` ⟨localization_list⟩

| ε

⟨localization⟩ ::= ⟨io_type⟩ ⟨lhside⟩ `:` ⟨address⟩

⟨io_type⟩ ::= `INPUT`

| `OUTPUT`

⟨address⟩ ::= ⟨nb⟩ `.` ⟨nb⟩ `.` ⟨nb⟩

⟨param_decl⟩ ::= ⟨n_empty_param_decl⟩

| ε

⟨n_empty_param_decl⟩ ::= ⟨id_list⟩ : ⟨type⟩ , ⟨n_empty_param_decl⟩

| ⟨id_list⟩ : ⟨type⟩

⟨id_list⟩ ::= ⟨id⟩

| ⟨id⟩ `,` ⟨id_list⟩

⟨type⟩ ::= `BOOL`
| `INT`
| `LONG`
| `REAL`
| ⟨id⟩

⟨block⟩ ::= ⟨instruction⟩ ; ⟨block⟩
| ε

⟨instruction⟩ ::= ⟨assign⟩
| ⟨wait⟩
| ⟨if⟩
| ⟨while⟩
| ⟨call⟩

⟨assign⟩ ::= ⟨lhside⟩ `:=` ⟨rhside⟩

⟨wait⟩ ::= `WAIT` ⟨rhside⟩

⟨if⟩ ::= `IF` ⟨rhside⟩ `THEN`
⟨block⟩
⟨else⟩

⟨else⟩ ::= `ELSE`
⟨block⟩
`END_IF`
| `END_IF`

⟨while⟩ ::= `WHILE` ⟨rhside⟩ `DO`
⟨block⟩
`END_WHILE`

⟨call⟩ ::= ⟨sequence_call⟩
| ⟨method_call⟩

⟨sequence_call⟩ ::= `LAUNCH` ⟨id⟩ `(` ⟨rhside_list⟩ `)`

⟨method_call⟩ ::= `LAUNCH` ⟨id⟩ `<-` ⟨id⟩ `(` ⟨rhside_list⟩ `)`
| ⟨id⟩ `<-` ⟨id⟩ `(` ⟨rhside_list⟩ `)`

⟨lhside⟩ ::= ⟨id⟩
| ⟨lhside⟩ . ⟨id⟩

⟨rhside⟩ ::= ⟨lhside⟩
| ∼ ⟨lhside⟩
| `(` ⟨rhside⟩ `)`
| ⟨un_op⟩ ⟨rhside⟩
| ⟨rhside⟩ ⟨bin_op⟩ ⟨rhside⟩
| ⟨constant⟩

⟨un_op⟩ ::= `NOT`
| −

⟨bin_op⟩ ::= `OR`
| `AND`
| `<`
| `>`
| `<=`
| `>=`
| `<>`
| `==`
| `%`
| `+`
| `-`
| `*`
| `/`

⟨constant⟩ ::= `TRUE`
| `FALSE`
| ⟨nb⟩

⟨rhside_list⟩ ::= ⟨n_empty_rhside_list⟩
| ε

⟨n_empty_rhside_list⟩ ::= ⟨rhside⟩ `,` ⟨n_empty_rhside_list⟩
| ⟨rhside⟩

A.2 Simplified Grammar

⟨dsl_program⟩ ::= ⟨decl_list⟩

⟨decl_list⟩ ::= ⟨decl⟩ ⟨decl_list⟩
| ε

⟨decl⟩ ::= ⟨var_decl⟩
| ⟨site_decl⟩
| ⟨event_decl⟩
| ⟨sequence_decl⟩

⟨var_decl⟩ ::= `VAR`
⟨var_list⟩
`END_VAR`

⟨site_decl⟩ ::= `SITE` ⟨id⟩
⟨localization_list⟩
`END_SITE`

⟨sequence_decl⟩ ::= `SEQUENCE` ⟨id⟩ `( )`
⟨var_decl⟩
⟨block⟩
`END_SEQUENCE`

⟨event_decl⟩ ::= `WHEN` ⟨rhside⟩ `THEN`
⟨block⟩
`END_WHEN`

⟨var_list⟩ ::= ⟨id_list⟩ `:` `BOOL` `;` ⟨var_list⟩
| ε

⟨localization_list⟩ ::= ⟨localization⟩ `;` ⟨localization_list⟩
| ε

⟨localization⟩ ::= ⟨io_type⟩ ⟨lhside⟩ `:` ⟨address⟩

⟨io_type⟩ ::= `INPUT`
| `OUTPUT`

⟨address⟩ ::= ⟨nb⟩ `.` ⟨nb⟩ `.` ⟨nb⟩

⟨id_list⟩ ::= ⟨id⟩
| ⟨id⟩ `,` ⟨id_list⟩

⟨block⟩ ::= ⟨instruction⟩ ; ⟨block⟩
| ε
⟨instruction⟩ ::= ⟨assign⟩
| ⟨wait⟩
| ⟨if⟩
| ⟨while⟩
| ⟨call⟩
⟨assign⟩ ::= ⟨lhside⟩ `:=` ⟨rhside⟩
⟨wait⟩ ::= `WAIT` ⟨rhside⟩
⟨if⟩ ::= `IF` ⟨rhside⟩ `THEN`
⟨block⟩
⟨else⟩
⟨else⟩ ::= `ELSE`
⟨block⟩
`END_IF`
⟨while⟩ ::= `WHILE` ⟨rhside⟩ `DO`
⟨block⟩
`END_WHILE`
⟨call⟩ ::= ⟨sequence_call⟩
⟨sequence_call⟩ ::= `LAUNCH` ⟨id⟩ `( )`
⟨lhside⟩ ::= ⟨id⟩
| ⟨lhside⟩ . ⟨id⟩
⟨rhside⟩ ::= ⟨lhside⟩
| $\sim$ ⟨lhside⟩
| ⟨constant⟩
| `(` ⟨rhside⟩ `)`
| ⟨un_op⟩ ⟨rhside⟩
| ⟨rhside⟩ ⟨bin_op⟩ ⟨rhside⟩
⟨constant⟩ ::= `TRUE`
| `FALSE`

⟨un_op⟩ ::= `NOT`

⟨bin_op⟩ ::= `OR`
| `AND`
| `<>`
| `==`

Appendix

B

One-Split Simulation Lemma

B.1 Lemma Statement

First, before turning our attention to the proof, let us recall the lemma statement.

Lemma 3.2 (One-Split Simulation Lemma)

Given a well formed dSL program $P = \langle V, E, S, s_0, \lhd, \Lambda \rangle$, and two distributions $D_1, D_2 \in \mathcal{D}_P$ such that $D_1 \preceq D_2$ and $|D_2| = |D_1| + 1$, we have that:

$$K_{P,D_1} \lesssim K_{P,D_2}$$

B.2 Preliminary Results

We shall need some preliminary results. The first one is about channel distribution. It stipulates that if a $\diamond$ marker is inserted in a communication channel σ, then $\diamond$ markers can also be inserted in the channels of any distribution $\langle \sigma_1, \sigma_2 \rangle$ of σ in such a way as to preserve the distribution.

Lemma B.1

Given the content of three communication channels $\sigma, \sigma_1, \sigma_2 \in (V \times \mathbb{B})^*$ without $\diamond$ markers such that $\sigma \prec \langle \sigma_1, \sigma_2 \rangle$, we have that :

$$\forall \sigma' \in (\sigma \parallel \diamond), \exists \sigma'_1 \in (\sigma_1 \parallel \diamond), \exists \sigma'_2 \in (\sigma_2 \parallel \diamond) : \sigma' \prec \langle \sigma'_1, \sigma'_2 \rangle$$

Proof

We know, by definition of $\parallel$, that $\sigma = \psi \cdot \psi'$ for some $\psi, \psi' \in (V \times \mathbb{B})^*$ such that $\sigma' = \psi \cdot \diamond \cdot \psi'$. We proceed by induction on $|\psi|$.

Proof (cont'd)

Initial Step If $\psi = \varepsilon$, we have that $\sigma = \psi'$. In this case, we can chose $\sigma'_1 = \diamond \cdot \sigma_1 \in (\sigma_1 \parallel \diamond)$ and $\sigma'_2 = \diamond \cdot \sigma_2 \in (\sigma_2 \parallel \diamond)$ and by Definition 3.11 of channel distribution, we have that $\sigma' = \diamond \cdot \sigma \prec \langle \diamond \cdot \sigma_1, \diamond \cdot \sigma_2 \rangle = \langle \sigma'_1, \sigma'_2 \rangle$.

Induction Step If $\psi = x \cdot \psi''$ with $x \in V \times \mathbb{B}$, since $\sigma \prec \langle \sigma_1, \sigma_2 \rangle$, we have by Definition 3.11 that either that $\sigma_1 = x \cdot \sigma''_1$ with $\psi'' \cdot \psi' \prec \langle \sigma''_1, \sigma_2 \rangle$ or that $\sigma_2 = x \cdot \sigma''_2$ with $\psi'' \cdot \psi' \prec \langle \sigma_1, \sigma''_2 \rangle$. In the former case, by induction, we have that there exists $\sigma'''_1 \in (\sigma''_1 \parallel \diamond)$ and $\sigma'''_2 \in (\sigma_2 \parallel \diamond)$ such that $\psi'' \cdot \diamond \cdot \psi' \prec \langle \sigma'''_1, \sigma'''_2 \rangle$. In this case, we can chose $\sigma'_1 = x \cdot \sigma'''_1 \in (x \cdot \sigma_1 \parallel \diamond)$ and $\sigma'_2 = \sigma'''_2 \in (\sigma_2 \parallel \diamond)$, and by Definition 3.9, we have that $\sigma' = x \cdot \psi'' \cdot \diamond \cdot \psi' \prec \langle x \cdot \sigma'''_1, \sigma'''_2 \rangle = \langle \sigma'_1, \sigma'_2 \rangle$. The latter case is symmetrical.

The second result is about blocks of code. It states that concatenation preserves their distribution.

Lemma B.2

Given blocks of code $\omega, \omega_1, \omega_2, \omega', \omega'_1, \omega'_2$. We have that:

$$(\omega \prec \langle \omega_1, \omega_2 \rangle) \wedge (\omega' \prec \langle \omega'_1, \omega'_2 \rangle) \Rightarrow (\omega; \omega' \prec \langle \omega_1; \omega'_1, \omega_2; \omega'_2 \rangle)$$

Proof

The proof is by induction on the length of ω.

Initial Step If $\omega = \varepsilon$, we have that $\omega \prec \langle \omega_1, \omega_2 \rangle$ implies by Definition 3.9 of code distribution that $\omega_1 = \omega_2 = \varepsilon$ which, in turns, implies $\omega; \omega' = \omega' \prec \langle \omega'_1, \omega'_2 \rangle = \langle \omega_1; \omega'_1, \omega_2; \omega'_2 \rangle$.

Induction Step If $\omega = x; \omega''$, there are two cases to consider:

(*i*) if $x \in \{\texttt{MSG}, \texttt{SEQ}\}$, Definition 3.9 implies that $(\omega_1 = x; \omega''_2) \wedge (\omega_1 = x; \omega''_1) \wedge (\omega'' \prec \langle \omega''_1, \omega''_2 \rangle)$. However, by induction, since $\omega'' \prec \langle \omega''_1, \omega''_2 \rangle$, we have that $\omega''; \omega' \prec \langle \omega''_1; \omega'_1, \omega''_2; \omega'_2 \rangle$. It follows by Definition 3.9, that $\omega; \omega' = x; \omega''; \omega' \prec \langle x; \omega''_1; \omega'_1, x; \omega''_2; \omega'_2 \rangle = \langle \omega_2; \omega'_2, \omega_1; \omega'_1 \rangle$.

(*ii*) if $x \notin \{\texttt{MSG}, \texttt{SEQ}\}$, Definition 3.9 implies that either $(\omega_1 = x; \omega''_1) \wedge (\omega'' \prec \langle \omega''_1, \omega_2 \rangle)$ or that $(\omega_2 = x; \omega''_2) \wedge (\omega'' \prec \langle \omega_1, \omega''_2 \rangle)$. In the former case, $(\omega'' \prec \langle \omega''_1, \omega_2 \rangle)$ implies, by induction that $\omega''; \omega \prec \langle \omega''_1; \omega_1, \omega_2; \omega_2 \rangle$. It follows, by Definition 3.9, that $\omega; \omega' = x; \omega''; \omega' \prec \langle x; \omega''_1; \omega'_1, \omega_2, \omega'_2 \rangle = \langle \omega_2; \omega'_2, \omega_1; \omega'_1 \rangle$. The latter case is symmetrical.

The third result is about the distribution of blocks of code corresponding to the sampling of inputs variables, the update of output variables and the treatment of events.

Lemma B.3

Given a well-formed dSL program $P = \langle V, E, S, s_0, \lhd \rangle$, two disjoint subsets $V_1, V_2 \subseteq V$, and two disjoint subsets $E_1, E_2 \subseteq E$, we have that:

(i) $\mathsf{in}(V_1 \cup V_2) \prec \langle \mathsf{in}(V_1), \mathsf{in}(V_2) \rangle$

(ii) $\mathsf{out}(V_1 \cup V_2) \prec \langle \mathsf{out}(V_1), \mathsf{out}(V_2) \rangle$

(iii) $\mathsf{treat}(E_1 \cup E_2) \prec \langle \mathsf{treat}(E_1), \mathsf{treat}(E_2) \rangle$

Proof

We prove (i) by induction on $|V_1 \cup V_2|$.

Initial step If $V_1 \cup V_2 = \varnothing$, By definition, we have that $\mathsf{in}(V_1 \cup V_2) = \varepsilon$. However, $V_1 \cup V_2 = \varnothing$ implies that $V_1 = V_2 = \varnothing$. Therefore, we have that $\mathsf{in}(V_1) = \mathsf{in}(V_2) = \varepsilon$. It follows by Definition 3.9 of code distribution that $\mathsf{in}(V_1 \cup V_2) \prec \langle \mathsf{in}(V_1), \mathsf{in}(V_2) \rangle$.

Induction step If $V_1 \cup V_2 \neq \varnothing$, let $x = \mathsf{min}_{\lhd}(V_1 \cup V_2)$. By definition, we have that $\mathsf{in}(V_1 \cup V_2) = \texttt{INPUT}(x); \mathsf{in}((V_1 \cup V_2) \setminus \{x\})$. Then, since V_1 and V_2 are disjoint, either $x \in V_1$ or $x \in V_2$. In the former case, $(V_1 \cup V_2) \setminus \{x\} = V_1 \setminus \{x\} \cup V_2$, and by induction, we have that $\mathsf{in}((V_1 \cup V_2) \setminus \{x\}) \prec \langle \mathsf{in}(V_1 \setminus \{x\}), \mathsf{in}(V_2) \rangle$. Finally, by Lemma B.2, since $\texttt{INPUT}(x) \prec \langle \texttt{INPUT}(x), \varepsilon \rangle$, we have that $\mathsf{in}(V_1 \cup V_2) = \texttt{INPUT}(x); \mathsf{in}((V_1 \cup V_2) \setminus \{x\}) \prec \langle \texttt{INPUT}(x); \mathsf{in}(V_1 \setminus \{x\}), V_2 \rangle = \langle \mathsf{in}(V_1), \mathsf{in}(V_2) \rangle$. The latter case ($x \in V_2$) is symmetrical.

A similar proof can be made for (ii) and (iii).

B.3 Proof of the Lemma

First, we assume that $S = \{s_1, s_2, ..., s_n\}$, $K_{P,D_1} = \langle Q_1, I_1, \mathcal{L}_1, \rightarrow_1 \rangle$, $K_{P,D_2} = \langle Q_2, I_2, \mathcal{L}_2, \rightarrow_2 \rangle$ and, without loss of generality, that $D_1 = \{X_{1,1}, X_{1,2}, ..., X_{1,k}\}$ and $D_2 = \{X_{2,1}, X_{2,2}, ...X_{2,k-1}, X_{2,k}, X_{2,k+1}\}$ with $\forall i \in [1, k) : X_{1,i} = X_{2,i}$ and $X_{1,k} = X_{2,k} \cup X_{2,k+1}$. Then, we define a relation $S \subseteq Q_1 \times Q_2$ such that two states

$$q_1 = \langle \langle \omega_{1,1}, \nu_{1,1}, \phi_{1,1} \rangle, ..., \langle \omega_{1,k}, \nu_{1,k}, \phi_{1,k} \rangle, \langle \sigma_{1,0}, \mu_{1,0} \rangle, ..., \langle \sigma_{1,n}, \mu_{1,n} \rangle \rangle$$
$$q_2 = \langle \langle \omega_{2,1}, \nu_{2,1}, \phi_{2,1} \rangle, ..., \langle \omega_{2,k+1}, \nu_{2,k+1}, \phi_{2,k+1} \rangle, \langle \sigma_{2,0}, \mu_{2,0} \rangle, ..., \langle \sigma_{2,n}, \mu_{2,n} \rangle \rangle$$

are related if and only if:

$$(i)\ \ (\forall i \in [1,k) : \omega_{1,i} = \omega_{2,i}) \wedge (\omega_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle)$$

$$(ii)\ \ (\forall i \in [1,k) : \nu_{1,i} = \nu_{2,i}) \wedge (\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle)$$

$$(iii)\ \ (a)\ \ \forall j \in [1,k), \forall i \in [1,k) : \phi_{1,i \leftarrow j} = \phi_{2,i \leftarrow j}$$

$$(b)\ \ \forall j \in [1,k] : \phi_{1,k \leftarrow j} = \phi_{2,k \leftarrow j} = \phi_{2,k+1 \leftarrow j}$$

$$(c)\ \ \forall i \in [1,k] : \phi_{1,i \leftarrow k} \prec \langle \phi_{2,i \leftarrow k}, \phi_{2,i \leftarrow k+1} \rangle$$

$$(iv)\ \ \forall j \in [0,n] : (\sigma_{1,j} = \sigma_{2,j}) \wedge (\mu_{1,j} = \mu_{2,j})$$

and prove that $K_{P,D_1} \trianglelefteq_S K_{P,D_2}$. First, from (ii) and Definition 3.10 of valuation distribution, we have that if $\langle q_1, q_2 \rangle \in S$ then $\mathcal{L}_1(q_1) = \mathcal{L}_2(q_2)$. Then, by Definition 3.8 of the semantics, we know that the set of initial states I_1 and I_2 are singletons. If $I_1 = \{q_{1,0}\}$ and $I_2 = \{q_{2,0}\}$, by definition of S, it is easy to see that $\langle q_{1,0}, q_{2,0} \rangle \in S$. Indeed, in those two states, all workloads and communication channels are empty, and all valuations are such that each variable is set to ff. Finally, it remains to prove that $\forall q_1' \in Q_1, \exists q_2', q_2'' \in Q_2$:

$$\big(q_1 \rightarrow_1 q_1'\big) \Rightarrow \big(\big(q_2 \overset{\sim}{\rightarrow}_2 q_2'' \rightarrow_2 q_2' \big) \wedge \big(\langle q_1', q_2' \rangle \in S \big) \big)$$

For that purpose, every rule of the structural operational semantics presented in Section 3.1.3 needs to be examined. For each of those rules, we show that if a transition can be derived in K_{P,D_1}, this transition can be simulated by some stuttering transitions, followed by one transition, all of which can be derived in K_{P,D_2}.

Cycle Start Rule If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.1) for some process i, we have that $\omega_{1,i} = \varepsilon$. There are two cases to consider:

(i) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \varepsilon$. Therefore, Rule (3.1) can be applied from q_2 leading to a state q_2' exactly like q_2 except for process i where

$$\omega_{2,i}' = \mathsf{in}(X_{2,i} \cap V_{in}); \mathsf{treat}(E_{X_{1,k}}); \mathtt{MSG}; \mathtt{SEQ}; \mathsf{out}(X_{2,i} \cap V_{out})$$

and $\nu_{2,i}' = \nu_{2,i}$ and where for all process $j \in [1, k+1]$ we have $\phi_{2,i \leftarrow j}' \in (\phi_{2,i \leftarrow j} \parallel \diamond)$. For the workload, since by hypothesis $X_{2,i} = X_{1,i}$, we have that $\omega_{2,i}' = \omega_{1,i}'$. For the valuation, by Definition of S, we have that $\nu_{2,i}' = \nu_{2,i} = \nu_{1,i}$. Finally, let us examine the communications channels. First, by definition of S, for all process $j \in [1,k)$, we have that $\phi_{1,i \leftarrow j} = \phi_{2,i \leftarrow j}$ which implies that $\phi_{2,i \leftarrow j}' \in$

$(\phi_{2,i\leftarrow j} \parallel \diamond) = (\phi_{1,i\leftarrow j} \parallel \diamond)$. Therefore, for those communication channels, we can choose to insert the $\diamond$ markers in $\phi_{2,i\leftarrow j}$ in the exact same way as in $\phi_{1,i\leftarrow j}$ to obtain $\phi'_{2,i\leftarrow j} = \phi'_{1,i\leftarrow j}$. Then, for process k, by the definition of S, we know that $\phi_{1,i\leftarrow k} \prec \langle \phi_{2,i\leftarrow k}, \phi_{2,i\leftarrow k+1} \rangle$ and by Lemma B.1, that there exists $\sigma_k \in (\phi_{2,i\leftarrow k} \parallel \diamond)$ and $\sigma_{k+1} \in (\phi_{2,i\leftarrow k+1} \parallel \diamond)$ such that $\phi'_{1,i\leftarrow k} \prec \langle \sigma_k, \sigma_{k+1} \rangle$. We can therefore choose $\phi'_{2,i\leftarrow k} = \sigma_k$ and $\phi'_{2,i\leftarrow k+1} = \sigma_{k+1}$, and finally conclude that $\langle q'_1, q'_2 \rangle \in S$.

(*ii*) If $i = k$, we have by definition of S that $\varepsilon = \omega_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$ and by Definition 3.9 of code distribution that $\omega_{2,k} = \omega_{2,k+1} = \varepsilon$. Therefore, we can first apply Rule (3.1) from q_2 for process k leading to a state q''_2. Note that Rule (3.1) does not change the valuations of the variables in V, which implies that $q_2 \overset{\sim}{\rightarrow}_{P,D_2} q''_2$. Then, from q''_2, since the workload of process $k+1$ is still empty, we can again apply Rule (3.1) (stuttering is needed for that), but this time for process $k+1$ leading to a state q'_2 exactly like q_2 except for processes k and $k+1$ where

$$\begin{array}{lcl} \omega'_{2,k} & = & \mathsf{in}(X_{2,k} \cap V_{in}); \mathsf{treat}(E_{X_{2,k}}); \texttt{MSG}; \texttt{SEQ}; \mathsf{out}(X_{2,k} \cap V_{out}) \\ \omega'_{2,k+1} & = & \mathsf{in}(X_{2,k+1} \cap V_{in}); \mathsf{treat}(E_{X_{2,k+1}}); \texttt{MSG}; \texttt{SEQ}; \mathsf{out}(X_{2,k+1} \cap V_{out}) \end{array}$$

and where $\nu'_{2,k} = \nu_{2,k}$ and $\nu'_{2,k+1} = \nu_{2,k+1}$ and where for all process $j \in [1, k+1]$, $\phi'_{2,k\leftarrow j} \in (\phi_{2,k\leftarrow j} \parallel \diamond)$ and $\phi_{2,k+1\leftarrow j} \in (\phi'_{2,k+1\leftarrow j} \parallel \diamond)$. First, let us examine the workloads. We know by hypothesis that $X_{1,k} = X_{2,k} \cup X_{2,k+1}$ and because $D_2 \in \mathcal{D}_P \subseteq \Pi(V)$ that $X_{2,k} \cap X_{2,k+1} = \varnothing$. By Lemma B.3, we have that

$$\begin{array}{lcl} \mathsf{in}(X_{1,k} \cap V_{in}) & \prec & \langle \mathsf{in}(X_{2,k} \cap V_{in}), \mathsf{in}(X_{2,k+1} \cap V_{in}) \rangle \\ \mathsf{out}(X_{1,k} \cap V_{in}) & \prec & \langle \mathsf{out}(X_{2,k} \cap V_{in}), \mathsf{out}(X_{2,k+1} \cap V_{in}) \rangle \end{array}$$

We also have, by definition, that $E_{X_{1,k}} = E_{X_{2,k}} \cup E_{X_{2,k+1}}$, and by Theorem 3.1 that $E_{X_{2,k}} \cap E_{X_{2,k+1}} = \varnothing$. This implies that

$$\mathsf{treat}(E_{X_{1,k}}) \prec \langle \mathsf{treat}(E_{X_{2,k}}), \mathsf{treat}(E_{X_{2,k+1}}) \rangle$$

Furthermore, by Definition 3.9, we have that $\texttt{MSG} \prec \langle \texttt{MSG}, \texttt{MSG} \rangle$ and that $\texttt{SEQ} \prec \langle \texttt{SEQ}, \texttt{SEQ} \rangle$. This implies, by Lemma B.2, that $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$. For the valuation, by definition of S, we have that $\nu'_{1,k} = \nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle = \langle \nu'_{2,k}, \nu'_{2,k+1} \rangle$. Finally, let us examine the communication channels. First, for all process $j \in [1, k)$, by definition of S, we have that $\phi_{1,k\leftarrow j} = \phi_{2,k\leftarrow j} = \phi_{2,k+1\leftarrow j}$, which implies that $\phi'_{2,k\leftarrow j} \in (\phi_{1,k\leftarrow j} \parallel \diamond)$ and that $\phi'_{2,k+1\leftarrow j} \in (\phi_{1,k\leftarrow j} \parallel \diamond)$. Therefore, for those communication channels, we can choose to insert the $\diamond$ markers in $\phi_{2,k\leftarrow j}$ and $\phi_{2,k+1\leftarrow j}$ in the exact same way as in $\phi_{1,k\leftarrow j}$ to obtain $\phi'_{2,k\leftarrow j} =$

$\phi'_{2,k+1\leftarrow j} = \phi'_{1,k\leftarrow j}$. For process k, by the definition of S, we know that $\phi_{1,k\leftarrow k} \prec \langle \phi_{2,k\leftarrow k}, \phi_{2,k\leftarrow k+1} \rangle$ and by Lemma B.1, that there exists $\sigma_k \in (\phi_{2,k\leftarrow k} \parallel \diamond)$ and $\sigma_{k+1} \in (\phi_{2,k\leftarrow k+1} \parallel \diamond)$ such that $\phi'_{1,k\leftarrow k} \prec \langle \sigma_k, \sigma_{k+1} \rangle$. We can therefore choose $\phi'_{2,k\leftarrow k} = \sigma_k$ and $\phi'_{2,k\leftarrow k+1} = \sigma_{k+1}$, and finally conclude that $\langle q'_1, q'_2 \rangle \in S$.

Input Rule If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.2) for some process i, we have that $\omega_{1,i} = \texttt{INPUT}(x); \omega'_{1,i}$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{INPUT}(x); \omega'_{1,i}$. Therefore, Rule (3.2) can be applied from state q_2, leading to a state q'_2 exactly like q_2 except for $\omega'_{2,i} = \omega'_{1,i}$ and $\nu'_{2,i} = \nu_{2,i}[x \mapsto a] = \nu_{1,i}[x \mapsto a] = \nu'_{1,i}$ for some $a \in \mathbb{B}$. It follows by Definition 3.9 and Definition 3.10 that $\langle q'_1, q'_2 \rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{INPUT}(x); \omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$ and by Definition 3.9 of code distribution that either $\omega_{2,k} = \texttt{INPUT}(x); \omega'_{2,k}$ with $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1} \rangle$, or $\omega_{2,k+1} = \texttt{INPUT}(x); \omega'_{2,k+1}$ with $\omega'_{1,k} \prec \langle \omega_{2,k}, \omega'_{2,k+1} \rangle$. In the former case, we can apply Rule (3.2) for process k from q_2, leading to a state q'_2 exactly like q_2, except for $\omega'_{2,k}$ defined above and for $\nu'_{2,k} = \nu_{2,k}[x \mapsto a]$. Finally, we have that $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1} \rangle = \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$, and by Definition 3.10 that $\nu'_{1,k} \prec \langle \nu'_{2,k}, \nu'_{2,k+1} \rangle$. We can therefore conclude that $\langle q'_1, q'_2 \rangle \in S$. The latter case is symmetrical.

Broadcast Rule If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.3) for some process i, we have that $\omega_{1,i} = \texttt{BCAST}(x, v); \omega'_{1,i}$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{BCAST}(x, v); \omega'_{1,i}$. Therefore, Rule (3.3) can be applied from state q_2, leading to a state q'_2 exactly like q_2 except for $\omega'_{2,i} = \omega'_{1,i}$ and $\phi'_{2,j\leftarrow i} = \phi_{2,j\leftarrow i} \cdot \langle x, v \rangle$ for all process $j \in [1, k+1]$. For all process $j \in [1, k)$, since $i < k$, by definition of S, we have that $\phi_{1,j\leftarrow i} = \phi_{2,j\leftarrow i}$. Therefore, for those communication channels, we have that $\phi'_{2,j\leftarrow i} = \phi_{2,j\leftarrow i} \cdot \langle x, v \rangle = \phi_{1,j\leftarrow i} \cdot \langle x, v \rangle = \phi'_{1,j\leftarrow i}$. Then, for process k, by definition of S, we know that $\phi_{1,k\leftarrow i} = \phi_{2,k\leftarrow i} = \phi_{2,k+1\leftarrow i}$. Therefore, we have that $\phi'_{2,k\leftarrow i} = \phi_{2,k\leftarrow i} \cdot \langle x, v \rangle = \phi_{1,k\leftarrow i} \cdot \langle x, v \rangle = \phi'_{1,k\leftarrow i}$ and that $\phi'_{2,k+1\leftarrow i} = \phi_{2,k+1\leftarrow i} \cdot \langle x, v \rangle = \phi_{1,k\leftarrow i} \cdot \langle x, v \rangle = \phi'_{1,k\leftarrow i}$. By Definition of S, we can conclude that $\langle q'_1, q'_2 \rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{BCAST}(x, v); \omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. By Definition 3.9, it follows that either $\omega_{2,k} = \texttt{BCAST}(x, v); \omega'_{2,k}$ with $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1} \rangle$, or $\omega_{2,k+1} = \texttt{BCAST}(x, v); \omega'_{2,k+1}$ with $\omega'_{1,k} \prec \langle \omega_{2,k}, \omega'_{2,k+1} \rangle$. In the

former case, we can apply Rule (3.3) for process k from q_2, leading to a state q_2' exactly like q_2, except for $\omega'_{2,k}$ defined above, and $\phi'_{2,j\leftarrow k} = \phi_{2,j\leftarrow k} \cdot \langle x, v\rangle$ for all process $j \in [1, k+1]$. For the workloads, we have that $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1}\rangle = \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$. Next, let us examine the communication channels. For all process $j \in [1, k)$, since $i < k$, by definition of S, and Definition 3.11, we have that $\phi'_{1,j\leftarrow k} = \phi_{1,j\leftarrow k} \cdot \langle x, v\rangle \prec \langle \phi_{2,j\leftarrow k} \cdot \langle x, v\rangle, \phi_{2,j\leftarrow k+1}\rangle = \langle \phi'_{2,j\leftarrow k}, \phi'_{2,,j\leftarrow k+1}\rangle$. For process k, by definition of S, we know that $\phi_{1,k\leftarrow k} = \phi_{2,k\leftarrow k} = \phi_{2,k+1\leftarrow k}$ and that $\phi_{1,k\leftarrow k} \prec \langle \phi_{2,k\leftarrow k}, \phi_{2,k\leftarrow k+1}\rangle$. It follows that $\phi'_{1,k\leftarrow k} = \phi_{1,k\leftarrow k} \cdot \langle x, v\rangle = \phi_{2,k\leftarrow k} \cdot \langle x, v\rangle = \phi'_{2,k\leftarrow k}$ and that $\phi'_{1,k\leftarrow k} = \phi_{1,k\leftarrow k} \cdot \langle x, v\rangle = \phi_{2,k+1\leftarrow k} \cdot \langle x, v\rangle = \phi'_{2,k+1\leftarrow k}$. It also follows, by Definition 3.11 that $\phi'_{1,k\leftarrow k} = \phi_{1,k\leftarrow k} \cdot \langle x, v\rangle \prec \langle \phi_{2,k\leftarrow k} \cdot \langle x, v\rangle, \phi_{2,k\leftarrow k+1}\rangle = \langle \phi'_{2,k\leftarrow k}, \phi'_{2,k\leftarrow k+1}\rangle$. By Definition of S, we can conclude that $\langle q_1', q_2'\rangle \in S$. The latter case is symmetrical.

Message Treatment Rules If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.4) for some process i, we have that $\omega_{1,i} = \mathtt{MSG}; \omega$ and $\phi_{1,i\leftarrow j} = \langle x, v\rangle \cdot \phi'_{1,i\leftarrow j}$ for some process $j \in [1, k]$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \mathtt{MSG}; \omega'_{1,i}$. Then, either $j < k$ and $\phi_{2,i\leftarrow j} = \phi_{1,i\leftarrow j} = \langle x, v\rangle \cdot \phi'_{1,i\leftarrow j}$, or $j = k$ and $\phi_{1,i\leftarrow k} = \langle x, v\rangle \cdot \phi'_{1,i\leftarrow k} \prec \langle \phi_{2,i\leftarrow k}, \phi_{2,i\leftarrow k+1}\rangle$, in which case, by Definition 3.11 of channel distribution, we have that $\phi_{2,i\leftarrow k} = \langle x, v\rangle \cdot \phi'_{2,i\leftarrow k}$ with $\phi'_{1,i\leftarrow k} \prec \langle \phi'_{2,i\leftarrow k}, \phi_{2,i\leftarrow k+1}\rangle$ or that $\phi_{2,i\leftarrow k+1} = \langle x, v\rangle \cdot \phi'_{2,i\leftarrow k+1}$ with $\phi'_{1,i\leftarrow k} \prec \langle \phi_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k+1}\rangle$. In any case, we have that $\exists \ell \in [1, k+1] : \phi_{2,i\leftarrow \ell} = \langle x, v\rangle \cdot \phi'_{2,i\leftarrow \ell}$. We can therefore apply Rule (3.4) from q_2 leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,i}); \omega$, for $\nu'_{2,i} = \nu_{2,i}[\tilde{x} \mapsto v]$ and for $\phi'_{2,i\leftarrow \ell}$ defined above. For the workloads, we know that $X_{2,i} = X_{1,i}$. This implies that $\omega'_{2,i} = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,i}); \omega = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{1,i}); \omega = \omega'_{1,i}$. For the valuation, by definition of S, we have that $\nu_{2,i} = \nu_{1,i}$, which implies that $\nu'_{2,i} = \nu_{2,i}[\tilde{x} \mapsto v] = \nu_{1,i}[\tilde{x} \mapsto v] = \nu'_{1,i}$. For the communication channels, either $\ell < k$ and $\phi'_{2,i\leftarrow \ell} = \phi'_{2,i\leftarrow j} = \phi'_{1,\leftarrow j}$, or $\ell = k$ and $\phi'_{1,i\leftarrow k} \prec \langle \phi'_{2,i\leftarrow k}, \phi_{2,i\leftarrow k+1}\rangle = \langle \phi'_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k+1}\rangle$, or $\ell = k+1$ and $\phi'_{1,i\leftarrow k} \prec \langle \phi_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k+1}\rangle = \langle \phi'_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k+1}\rangle$. In any case, by definition of S, we have that $\langle q_1', q_2'\rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \mathtt{MSG}; \omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$. By Definition 3.9 of code distribution, it follows that $\omega_{2,k} = \mathtt{MSG}; \omega'$ and that $\omega_{2,k+1} = \mathtt{MSG}; \omega''$ with $\omega \prec \langle \omega', \omega''\rangle$. Moreover, again by definition of S, we have that $\phi_{2,k\leftarrow j} = \phi_{2,k+1\leftarrow j} = \phi_{1,k\leftarrow j} = \langle x, v\rangle \cdot \phi'_{1,k\leftarrow j}$. Therefore, we can

first apply Rule (3.4) from q_2 for process k, leading to a state q_2''. Note that Rule (3.4) does not change the valuations of the variables in V, which implies that $q_2 \overset{\sim}{\rightarrow}_{P,D_2} q_2''$. Then, from q_2'', since the workload of process $k+1$ still begins with $\texttt{MSG}$, we can apply Rule (3.4), but this time for process $k+1$ (stuttering is needed for that), leading to a state q_2' exactly like q_2 except for $\omega_{2,k} = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k}); \omega'$ and $\omega_{2,k+1} = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k+1}); \omega''$, for $\nu'_{2,k} = \nu_{2,k}[\tilde{x} \mapsto v]$ and $\nu'_{2,k+1} = \nu_{2,k+1}[\tilde{x} \mapsto v]$ and for $\phi_{2,k\leftarrow k}$ and $\phi_{2,k+1\leftarrow j}$ defined above. For the workloads, we know that $X_{1,k} = X_{2,k} \cup X_{2,k+1}$, which implies that $\widetilde{\mathsf{evt}}(x) \cap X_{1,k} = \widetilde{\mathsf{evt}}(x) \cap (X_{2,k} \cup X_{2,k+1}) = (\widetilde{\mathsf{evt}}(x) \cap X_{2,k}) \cup (\widetilde{\mathsf{evt}}(x) \cap X_{2,k+1})$. Furthermore, since $D_2 \in \mathcal{D}_P \subseteq \Pi(V)$, we have that $X_{2,k} \cap X_{2,k+1} = \varnothing$, which implies that we have that $(\widetilde{\mathsf{evt}}(x) \cap X_{2,k}) \cap (\widetilde{\mathsf{evt}}(x) \cap X_{2,k+1}) = \varnothing$. Therefore, by Lemma B.3, $\mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{1,k}) \prec \langle \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k}), \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k+1}) \rangle$. We also know that, $\omega \prec \langle \omega', \omega'' \rangle$, which implies, by Lemma B.2, that $\omega'_{1,k} = \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{1,k}); \omega \prec \langle \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k}); \omega', \mathsf{treat}(\widetilde{\mathsf{evt}}(x) \cap X_{2,k+1}); \omega'' \rangle = \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$. For the valuation, by definition of S, we have that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$. It follows by Definition 3.10 of valuation distribution, that $\nu'_{1,k} = \nu_{1,k}[\tilde{x} \mapsto v] \prec \langle \nu_{2,k}[\tilde{x} \mapsto v], \nu_{2,k+1}[\tilde{x} \mapsto v] \rangle = \langle \nu'_{2,k}, \nu'_{2,k+1} \rangle$. For the communication channels, we have that $\phi'_{2,k\leftarrow j} = \phi'_{2,k+1\leftarrow j} = \phi_{1,k'\leftarrow j}$. We can therefore conclude that $\langle q'_1, q'_2 \rangle \in S$.

End of Message Treatment Rule If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.5) for some process i, we have that $\omega_{1,i} = \texttt{MSG}; \omega'_{1,i}$ and that $\forall j \in [1,k] : \phi_{1,i\leftarrow j} = \diamond \cdot \phi'_{1,i\leftarrow j}$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have have $\omega_{2,i} = \omega_{1,i} = \texttt{MSG}; \omega_{1,i}$. Moreover, again by definition of S, we have that $\forall j \in [1,k) : \phi_{1,i\leftarrow j} = \phi_{2,i\leftarrow j}$ and that $\phi_{1,i\leftarrow k} \prec \langle \phi_{2,i\leftarrow k}, \phi_{2,i\leftarrow k} \rangle$. By Definition 3.11, it follows that $\phi_{2,i\leftarrow k} = \diamond \cdot \phi'_{2,i\leftarrow k}$ and $\phi_{2,i\leftarrow k+1} = \diamond \cdot \phi'_{2,i\leftarrow k+1}$ with $\phi'_{1,i\leftarrow k} \prec \langle \phi'_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k} \rangle$. Therefore, since there are $\diamond$ markers at the beginning of every receiving communication channels of process i in q_2, Rule (3.5) can be applied from q_2, leading to a state q'_2 exactly like q_2 except for $\omega'_{2,i}$ and $\phi'_{2,i}$ defined above. Finally, since $\omega'_{2,i} = \omega'_{1,i}$ and $\phi'_{1,i\leftarrow k} \prec \langle \phi'_{2,i\leftarrow k}, \phi'_{2,i\leftarrow k} \rangle$, by definition of S, we have that $\langle q'_1, q'_2 \rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,i} = \texttt{MSG}; \omega'_{i,1} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. By Definition 3.9 of code distribution, we have that $\omega_{2,k} = \texttt{MSG}; \omega'_{2,k}$ and that $\omega_{2,k+1} = \texttt{MSG}; \omega'_{2,k+1}$ with $\omega'_{1,k} \prec \omega'_{2,k}, \omega'_{2,k+1}$. Moreover, again by definition of S, for all process $j \in [1,k]$, we have that $\phi_{2,k\leftarrow j} = \phi_{1,k\leftarrow j} = \diamond \cdot \phi'_{1,k\leftarrow j}$ and that $\phi_{2,k+1\leftarrow j} = \phi_{1,k\leftarrow j} = \diamond \cdot \phi'_{1,k\leftarrow j}$. Therefore, we can first apply Rule (3.5) from

q_2 for process k leading to a state q_2''. Note that Rule (3.5) does not change the valuations of the variables in V, which implies that $q_2 \overset{\sim}{\rightarrow}_{P,D_2} q_2''$. Then, from q_2'', since the workload of process $k+1$ still begins with `MSG`, we can again apply Rule (3.5), but this time for process $k+1$ (stuttering is needed for that) leading to a state q_2' exactly like q_2 except for $\omega'_{2,k}$ and $\omega'_{2,k+1}$ defined above, and for $\phi'_{2,k} = \phi'_{1,k}$ and $\phi'_{2,k+1} = \phi'_{1,i}$. Moreover, we know by definition of S, that $\phi_{1,k\leftarrow k} = \diamond \cdot \phi'_{1,k\leftarrow k} \prec \langle \phi_{2,k\leftarrow k}, \phi_{2,k\leftarrow k+1}\rangle = \langle \diamond \cdot \phi_{2,k\leftarrow k}, \diamond \cdot \phi_{2,k\leftarrow k+1}\rangle$. It follows, by Definition 3.11 of channel distribution that $\phi'_{1,k\leftarrow k} \prec \langle \phi'_{2,k\leftarrow k}, \phi'_{2,k\leftarrow k+1}\rangle$. We can therefore conclude, by definition of S that $\langle q_1', q_2'\rangle \in S$.

Output Rule If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.6) for some process i, we have that $\omega_{1,i} = \texttt{OUTPUT}(x); \omega'_{1,i}$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{OUTPUT}(x); \omega'_{1,i}$. Therefore, Rule (3.6) can be applied from state q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = \omega'_{1,i}$ and $\nu'_{2,i} = \nu_{2,i}[x \mapsto \nu_{2,i}(\hat{x})]$. By definition of S, we have that $\nu_{1,i} = \nu_{2,i}$, therefore $\nu'_{2,i} = \nu_{2,i}[x \mapsto \nu_{2,i}(\hat{x})] = \nu_{1,i}[x \mapsto \nu_{1,i}(\hat{x})] = \nu'_{1,i}$. It follows by Definition 3.9 and Definition 3.10 that $\langle q_1', q_2'\rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{OUTPUT}(x); \omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$ and by Definition 3.9 of code distribution that either $\omega_{2,k} = \texttt{OUTPUT}(x); \omega'_{2,k}$ with $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1}\rangle$, or $\omega_{2,k+1} = \texttt{OUTPUT}(x); \omega'_{2,k+1}$ with $\omega'_{1,k} \prec \langle \omega_{2,k}, \omega'_{2,k+1}\rangle$. In the former case, we can apply Rule (3.6) for process k from q_2, leading to a state q_2' exactly like q_2, except for $\omega'_{2,k}$ defined above and $\nu'_{2,k} = \nu_{2,k}[x \mapsto \nu_{2,k}(\hat{x})]$. By definition of S, we have that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1}\rangle$, which implies that $\nu_{2,k}(\hat{x}) = \nu_{1,k}(\hat{x})$. It follows Definition 3.9 that $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$ and by Definition 3.10 that $\nu'_{1,k} \prec \langle \nu'_{2,k}, \nu'_{2,k+1}\rangle$. In this case, we therefore have $\langle q_1', q_2'\rangle \in S$. The latter case is symmetrical.

Assignment Rules If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.7), Rule (3.8) or Rule (3.9), for some process i, we have that $\omega_{1,i} = x := v; \omega$. First let us examine Rule (3.7). There are two cases to consider:

(*i*) If $i < k$, then by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = x := v; \omega$ and $\nu_{1,i} = \nu_{2,i}$. Moreover, since Rule (3.7) was applied from q_1, it must be that $x \in \{c_e \mid e \in E\}$. Therefore, Rule (3.7) can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = \omega = \omega_{1,i}$ and for $\nu'_{2,i} = \nu_{2,i}[x \mapsto \mathsf{eval}[\nu_{2,i}](v)] = \nu_{1,i}[x \mapsto \mathsf{eval}[\nu_{1,i}](v)] = \nu'_{1,i}$. We can therefore conclude, that $\langle q_1', q_2'\rangle \in S$.

(*ii*) If $i = k$, then by definition of S, we have that $\omega_{1,k} = x := v; \omega \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. Moreover, since Rule (3.7) was applied from q_1, it must be that $x \in \{c_e \mid e \in E\}$. By Definition 3.9, we have that either $\omega_{2,k} = x := v; \omega'$ with $\omega \prec \langle \omega', \omega_{2,k+1} \rangle$ or $\omega_{2,k} = x := v; \omega'$ with $\omega \prec \langle \omega_{2,k}, \omega' \rangle$. In the former case, Rule (3.7) can be applied from q_2 for process k, leading to a state q_2' exactly like q_2 except for $\omega_{2,k} = \omega'$ and $\nu'_{2,i} = \nu_{2,i}[x \mapsto \mathsf{eval}[\nu_{2,i}](v)]$. For the workloads, we have that $\omega'_{1,k} = \omega \prec \langle \omega', \omega_{2,k+1} \rangle = \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$. For the valuations, we have that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$, which implies by Definition 3.10 that $\mathsf{eval}[\nu_{2,i}](v) = \mathsf{eval}[\nu_{1,i}](v)$. It follows that $\nu'_{2,i}(x) = \nu'_{1,k}(x)$. Then, since x was assigned in $\omega_{2,k}$, we know that $x \in X_{2,k}$. Therefore, since $X_{2,k} \cap X_{2,k+1} = \varnothing$, we know that x does not belong to the domain of $\nu_{2,k+1}$, and we have that $\nu'_{1,i} \prec \langle \nu'_{2,k}, \nu_{2,k+1} \rangle = \langle \nu'_{2,k}, \nu'_{2,k+1} \rangle$. We can therefore conclude that $\langle q_1', q_2' \rangle \in S$. The latter case is symmetrical.

Then let us consider Rule (3.8) and Rule (3.9). Again, there are two cases to consider:

(*i*) If $i < k$, then by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = x := v; \omega$ and $\nu_{1,i} = \nu_{2,i}$. Moreover, since Rule (3.8) or Rule (3.9) was applied from q_1, it must be that either $x \in X_i \cap V_{out}$ or $x \in X_i \setminus V_{out}$. In each case, the same rule can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = \mathtt{BCAST}(x, \mathsf{eval}[\nu_{2,i}](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega = \mathtt{BCAST}(x, \mathsf{eval}[\nu_{1,i}](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega = \omega'_{1,i}$ and for $\nu'_{2,i}$. For the valuation, either $x \in V_{out}$, in which case $\nu'_{2,i} = \nu_{2,i}[\hat{x} \mapsto \mathsf{eval}[\nu_{2,i}](v)] = \nu_{1,i}[\hat{x} \mapsto \mathsf{eval}[\nu_{1,i}](v)] = \nu'_{1,i}$, or $x \notin V_{out}$, in which case $\nu'_{2,i} = \nu_{2,i}[x \mapsto \mathsf{eval}[\nu_{2,i}](v)] = \nu_{1,i}[x \mapsto \mathsf{eval}[\nu_{1,i}](v)] = \nu'_{1,i}$. In both case, by definition of S, we have that $\langle q_1', q_2' \rangle \in S$.

(*ii*) If $i = k$, then by definition of S, we have that $\omega_{1,k} = x := v; \omega \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. Moreover, since Rule (3.8) or Rule (3.9) was applied from q_1, it must be that either $x \in X_i \cap V_{out}$ or $x \in X_i \setminus V_{out}$. By Definition 3.9, we also have that either $\omega_{2,k} = x := v; \omega'$ with $\omega \prec \langle \omega', \omega_{2,k+1} \rangle$ or $\omega_{2,k} = x := v; \omega'$ with $\omega \prec \langle \omega_{2,k}, \omega' \rangle$. In the former case, the same rule can be applied from q_2 for process k, leading to a state q_2' exactly like q_2 except for $\omega'_{2,k} = \mathtt{BCAST}(x, \mathsf{eval}[\nu_{2,k}](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega'$ and for $\nu'_{2,k}$. For the valuations, we have that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$ by definition of S, which implies that $\mathsf{eval}[\nu_{2,k}](v) = \mathsf{eval}[\nu_{1,k}](v)$. It follows that $\nu'_{1,k}(x) = \nu'_{2,k}(x)$. Then, since x was assigned in $\omega_{2,k}$, we know that $x \in X_{2,k}$. Therefore, since $X_{2,k} \cap X_{2,k+1} = \varnothing$, we know that x does not belong to the domain of $\nu_{2,k+1}$, and we have that $\nu'_{1,i} \prec \langle \nu'_{2,k}, \nu_{2,k+1} \rangle = \langle \nu'_{2,k}, \nu'_{2,k+1} \rangle$. For the workloads, since $\omega \prec \langle \omega', \omega_{2,k+1} \rangle$, and since $\mathsf{eval}[\nu_{2,k}](v) = \mathsf{eval}[\nu_{1,k}](v)$, we have that $\omega'_{1,i} = \mathtt{BCAST}(x, \mathsf{eval}[\nu_{1,k}](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega \prec \langle \mathtt{BCAST}(x, \mathsf{eval}[\nu_{2,k}](v)); \mathsf{treat}(\mathsf{evt}(x)); \omega,$

$\omega_{2,k+1}\rangle = \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$. We can therefore conclude that $\langle q'_1, q'_2\rangle \in S$. The latter case is symmetrical.

If Rules If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.10) or Rule (3.11), for some process i, we have that $\omega_{1,i} = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF};\omega$. There are two cases to consider:

(i) If $i < k$, then by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF};\omega$ and that $\nu_{1,i} = \nu_{2,i}$. Then, either $\mathsf{eval}[\nu_{1,i}](e) = \mathsf{eval}[\nu_{2,i}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{1,i}](e) = \mathsf{eval}[\nu_{2,i}](e) = \mathsf{ff}$. In the case $\mathsf{eval}[\nu_{2,i}](e) = \mathsf{tt}$, Rule (3.10) can by applied from q_2 leading to a state q'_2 exactly like q_2 except for $\omega'_{2,i} = \omega_{\texttt{THEN}};\omega = \omega'_{1,i}$. In this case, by Definition of S, we have that $\langle q'_1, q'_2\rangle \in S$. In the case $\mathsf{eval}[\nu_{2,i}](e) = \mathsf{ff}$, Rule (3.11) can be applied instead of Rule (3.10) and the same proof can be made with $\omega_{\texttt{ELSE}}$ instead of $\omega_{\texttt{THEN}}$.

(ii) If $i = k$, then by definition of S, we have that $\omega_{1,k} = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF};\omega \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$ and that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1}\rangle$. By Definition 3.9, we have that either $\omega_{2,k} = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \ \omega_{\texttt{ELSE}}\ \texttt{END\_IF};\omega'$ with $\omega \prec \langle \omega', \omega_{2,k+1}\rangle$, or $\omega_{2,k+1} = \texttt{IF}\ e\ \texttt{THEN}\ \omega_{\texttt{THEN}}\ \texttt{ELSE}\ \omega_{\texttt{ELSE}}\ \texttt{END\_IF};\omega'$ and $\omega \prec \langle \omega_{2,k}, \omega'\rangle$. In the former case, we have that either $\mathsf{eval}[\nu_{1,k}](e) = \mathsf{eval}[\nu_{2,k}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{1,k}](e) = \mathsf{eval}[\nu_{2,k}](e) = \mathsf{ff}$. In the case, $\mathsf{eval}[\nu_{2,k}](e) = \mathsf{tt}$, Rule (3.10) can be applied for process k from state q_2, leading to a state q'_2 exactly like q_2 except for $\omega'_{2,k} = \omega_{\texttt{THEN}};\omega'$. Then, since $\omega_{\texttt{THEN}} \prec \langle \omega_{\texttt{THEN}}, \varepsilon\rangle$ and since $\omega \prec \langle \omega', \omega_{2,k+1}\rangle = \langle \omega', \omega'_{2,k+1}\rangle$, by Lemma B.2, we have that $\omega'_{1,k} = \omega_{\texttt{THEN}};\omega \prec \langle \omega_{\texttt{THEN}};\omega', \omega'_{2,k+1}\rangle = \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$. Finally, by definition of S, we can conclude that $\langle q'_1, q'_2\rangle \in S$. Still in this former case, if $\mathsf{eval}[\nu_{2,k}](e) = \mathsf{ff}$, Rule (3.11) can be applied instead of Rule (3.10) and the same proof can be made with $\omega_{\texttt{ELSE}}$ instead of $\omega_{\texttt{THEN}}$. The latter case, is symmetrical.

Launch Rules If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.12) or Rule (3.13), for some process i, we have that $\omega_{1,i} = \texttt{LAUNCH}\, s_j;\omega'_{1,i}$. First note, by definition of S, that $\sigma_{1,j} = \sigma_{2,j}$. Then, we have to examine two cases:

(i) If $i < k$, then by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{LAUNCH}\, s_j;\omega'_{1,i}$. In this case, either $\sigma_{2,j} = \sigma_{1,j} = \varepsilon$ and Rule (3.12) can be applied, or $\sigma_{2,j} = \sigma_{1,j} \neq \varepsilon$ and Rule (3.13) can be applied. In both cases, a transition can be applied from state q_2, leading to a state q'_2 exactly like q_2 except from $\omega'_{2,i}$, and if Rule (3.12) was used, for $\sigma'_{2,j}$ and $\mu'_{2,j}$. For the workloads, by definition of S, we have that

$\omega'_{2,i} = \omega'_{1,i}$. For the local state of sequence s_j, either Rule (3.12) was used, and $\sigma'_{2,j} = \mathsf{body}(s_j) = \sigma'_{1,j}$ with $\mu'_{2,j} = \mu_{2,j}[\mathsf{local}(s_j) \mapsto \mathsf{ff}] = \mu_{1,j}[\mathsf{local}(s_j) \mapsto \mathsf{ff}] = \mu'_{1,j}$, or Rule (3.13) was used and $\omega'_{2,i} = \omega_{2,i} = \omega_{1,i} = \omega'_{1,i}$ with $\mu'_{2,i} = \mu_{2,i} = \mu_{1,i} = \mu'_{1,i}$. In both cases, by definition of S, we have that $\langle q'_1, q'_2 \rangle \in S$.

(*ii*) If $i = k$, then we have that $\omega_{1,k} = \texttt{LAUNCH}\, s_j; \omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$ by definition of S. By Definition 3.9, we have that either $\omega_{2,k} = \texttt{LAUNCH}\, s_j; \omega'_{2,k}$ with $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1} \rangle$, or $\omega_{2,k+1} = \texttt{LAUNCH}\, s_j; \omega'_{2,k+1}$ with $\omega'_{1,k} \prec \langle \omega_{2,k}, \omega'_{2,k+1} \rangle$. In the former case, either $\sigma_{2,j} = \sigma_{1,j} = \varepsilon$ and Rule (3.12) can be applied for process k, or $\sigma_{2,j} = \sigma_{1,j} \neq \varepsilon$ and Rule (3.13) can be applied for process k. In both cases, a transition can be applied from state q_2, leading to a state q'_2 exactly like q_2 except from $\omega'_{2,k}$, and if Rule (3.12) was used, for $\sigma'_{2,j}$ and $\mu'_{2,j}$. For the workloads, we have that $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega_{2,k+1} \rangle = \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$. For the local state of sequence s_j, either Rule (3.12) was used, and $\sigma'_{2,j} = \mathsf{body}(s_j) = \sigma'_{1,j}$ with $\mu'_{2,j} = \mu_{2,j}[\mathsf{local}(s_j) \mapsto \mathsf{ff}] = \mu_{1,j}[\mathsf{local}(s_j) \mapsto \mathsf{ff}] = \mu'_{1,j}$, or Rule (3.12) was used and $\omega'_{2,i} = \omega_{2,i} = \omega_{1,i} = \omega'_{1,i}$ with $\mu'_{2,i} = \mu_{2,i} = \mu_{1,i} = \mu'_{1,i}$. In both cases, by definition of S, we have that $\langle q'_1, q'_2 \rangle \in S$. The latter case is symmetrical.

Sequence Assignment Rules If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.14) or Rule (3.15) for some process i, we have that $\omega_{1,i} = \texttt{SEQ}; \omega$ and that $\sigma_{1,j} = x := e; \sigma'_{1,j}$. If $x \in \mathsf{local}(s_j)$, the transition was derived using Rule (3.14). In this case, we have to consider two cases:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ}; \omega$ and $\sigma_{2,j} = \sigma_{1,j} = x := e; \sigma'_{1,j}$. Since Rule (3.14) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,i} \cup \mathsf{local}(s_j) = X_{2,i} \cup \mathsf{local}(s_j)$. Therefore, we can apply Rule (3.14) from state q_2, leading to a state q'_2 exactly like q_2 except for $\sigma'_{2,j} = \sigma'_{1,j}$ and for $\mu'_{2,j} = \mu_{2,j}[x \mapsto \mathsf{eval}[\mu_{2,j} \cup \nu_{2,i}](e)]$. By definition of S, we know that $\mu_{2,j} = \mu_{1,j}$ and that $\nu_{2,i} = \nu_{1,i}$. It follows that $\mu'_{2,j} = \mu_{1,j}[x \mapsto \mathsf{eval}[\mu_{1,j} \cup \nu_{1,i}](e)] = \mu'_{1,j}$. Therefore, by definition of S, we have that $\langle q'_1, q'_2 \rangle \in Q$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{SEQ}; \omega \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$ and $\sigma_{2,j} = \sigma_{1,j} = x := e; \sigma'_{1,j}$. First, by Definition 3.9 of code distribution, we have that $\omega_{2,k} = \texttt{SEQ}; \omega'$ and $\omega_{2,k+1} = \texttt{SEQ}; \omega''$ with $\omega \prec \langle \omega', \omega'' \rangle$. Then, since Rule (3.14) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,k} \cup \mathsf{local}(s_j) = X_{2,k} \cup X_{2,k+1} \cup \mathsf{local}(e)$. However, since $D_2 \in \mathcal{D}_P$, by Definition 3.3 of distribution, we have that $\exists X_{2,\ell} \in D_2 : \mathsf{var}(x := e) \subseteq X_{2,\ell} \cap \mathsf{local}(s_j)$. It follows that either $\ell = k$ or $\ell = k+1$. In the former case, we can apply Rule (3.14) for process k, from

state q_2, leading to a state q_2' exactly like q_2 except for $\sigma'_{2,j} = \sigma'_{1,j}$ and for $\mu'_{2,j} = \mu_{2,j}[x \mapsto \mathsf{eval}[\mu_{2,j} \cup \nu_{2,k}](e)]$. By definition of S, we know that $\mu_{2,j} = \mu_{1,j}$ and that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$. By Definition 3.10, we have that $\mathsf{eval}[\mu_{2,j} \cup \nu_{2,k}](e) = \mathsf{eval}[\mu_{1,j} \cup \nu_{1,k}](e)$ which implies that $\mu'_{2,j} = \mu_{1,j}[x \mapsto \mathsf{eval}[\mu_{1,j} \cup \nu_{1,k}](e)] = \mu'_{1,k}$. Therefore, by definition of S, we have that $\langle q_1', q_2' \rangle \in Q$. The latter case is symmetrical.

On the other hand, if $x \in X_{1,i}$, the transition was derived using Rule (3.15). In this case, we also have to consider two cases:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ}; \omega$ and $\sigma_{2,j} = \sigma_{1,j} = x := e; \sigma'_{1,j}$. Since Rule (3.15) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,i} \cup \mathsf{local}(s_j) = X_{2,i} \cup \mathsf{local}(s_j)$. Therefore, we can apply Rule (3.15) from state q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = x := \mathsf{eval}[\mu_{2,j} \cup \nu_{2,i}](e); \omega_{2,i}$ and for $\sigma'_{2,j} = \sigma'_{1,j}$. For the workloads, by definition of S, we have that $\omega_{1,i} = \omega_{2,i}$, that $\nu_{1,i} = \nu_{2,i}$ and that $\mu_{1,j} = \mu_{2,j}$. Therefore, $\omega'_{2,i} = x := \mathsf{eval}[\mu_{2,j} \cup \nu_{2,i}](e); \omega_{2,i} = x := \mathsf{eval}[\mu_{1,j} \cup \nu_{1,i}](e); \omega_{1,i} = \omega'_{1,i}$. We can therefore conclude that $\langle q_1', q_2' \rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{SEQ}; \omega \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$ and $\sigma_{2,j} = \sigma_{1,j} = x := e; \sigma'_{1,j}$. First, by Definition 3.9 of code distribution, we have that $\omega_{2,k} = \texttt{SEQ}; \omega'$ and $\omega_{2,k+1} = \texttt{SEQ}; \omega''$ with $\omega \prec \langle \omega', \omega'' \rangle$. Then, since Rule (3.15) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,k} \cup \mathsf{local}(s_j) = X_{2,k} \cup X_{2,k+1} \cup \mathsf{local}(e)$. However, since $D_2 \in \mathcal{D}_P$, by Definition 3.3 of distribution, we have that $\exists X_{2,\ell} \in D_2 : \mathsf{var}(x := e) \subseteq X_{2,\ell} \cap \mathsf{local}(s_j)$. It follows that either $\ell = k$ or $\ell = k + 1$. In the former case, we can apply Rule (3.15) for process k, from state q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,k} = x := \mathsf{eval}[\nu_{2,i} \cup \mu_{2,j}](e); \omega_{2,k}$ and for $\mu'_{2,j} = \mu'_{1,j}$. For the workloads, by definition of S, we have that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$ and that $\mu_{2,j} = \mu_{1,j}$, which implies, by Definition 3.10 of valuation distribution, that $\mathsf{eval}[\nu_{2,k} \cup \mu_{2,j}](e) = \mathsf{eval}[\nu_{1,k} \cup \mu_{1,j}](e)$. It follows that $x := \mathsf{eval}[\nu_{1,k} \cup \mu_{1,j}](e) \prec \langle x := \mathsf{eval}[\nu_{2,k} \cup \mu_{2,j}](e), \varepsilon \rangle$. We also know that $\omega_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. This implies by Lemma B.2, that $\omega'_{1,k} = x := \mathsf{eval}[\nu_{1,k} \cup \mu_{1,j}](e); \omega_{1,k} \prec \langle x := \mathsf{eval}[\nu_{2,k} \cup \mu_{2,j}](e); \omega_{2,k}, \omega_{2,k+1} \rangle = \langle \omega'_{2,k}, \omega'_{2,k+1} \rangle$. Therefore, we can conclude that $\langle q_1', q_2' \rangle \in S$.

Sequence If Rules If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.16) or Rule (3.17) for some process i, we have that $\omega_{1,i} = \texttt{SEQ}; \omega$ and that $\sigma_{1,j} = \texttt{IF}\, e\, \texttt{THEN}\, \sigma_{\texttt{THEN}}\, \texttt{ELSE}\, \sigma_{\texttt{ELSE}}\, \texttt{END\_IF}; \sigma$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ};\omega$, that $\nu_{2,i} = \nu_{1,i}$, that $\sigma_{2,j} = \sigma_{1,j} = \texttt{IF}\, e\, \texttt{THEN}\, \sigma_{\texttt{THEN}}\, \texttt{ELSE}\, \sigma_{\texttt{ELSE}}\, \texttt{END\_IF};\sigma$, and that $\mu_{2,j} = \mu_{1,j}$. Since either Rule (3.16) or Rule (3.17) was applied from q_2, it must be that $\mathsf{var}(e) \subseteq X_{1,i} \cup \mathsf{local}(s_j) =\subseteq X_{2,i} \cup \mathsf{local}(s_j)$. Then, either $\mathsf{eval}[\nu_{1,j} \cup \mu_{1,i}](e) = \mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{1,j} \cup \mu_{1,i}](e) = \mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{ff}$. In the case $\mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{tt}$, Rule (3.16) can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\sigma'_{2,j} = \sigma_{\texttt{THEN}};\sigma = \sigma_{1,j}$. In this case, by definition of S, we have that $\langle q_1', q_2'\rangle \in S$. In the case $\mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{tt}$, Rule (3.17) can be applied instead of Rule (3.16) and the same proof can be made, with $\sigma_{\texttt{ELSE}}$ instead $\sigma_{\texttt{THEN}}$.

(*ii*) If $i = k$, by definition of S, we have that $\sigma_{2,j} = \sigma_{1,j} = \texttt{IF}\, e\, \texttt{THEN}\, \sigma_{\texttt{THEN}}\ \texttt{ELSE}\ \sigma_{\texttt{ELSE}}\ \texttt{END\_IF};\sigma$ and that $\omega_{1,k} = \texttt{SEQ};\omega \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$, which implies, by Definition 3.9 that $\omega_{2,k} = \texttt{SEQ};\omega'$ and $\omega_{2,k+1} = \texttt{SEQ};\omega''$ with $\omega \prec \langle \omega', \omega''\rangle$. We also have by definition of S that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1}\rangle$ and that $\mu_{1,j} = \mu_{2,j}$. Then, since Rule (3.16) of Rule (3.17) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,k} \cup \mathsf{local}(s_j) = X_{2,k} \cup X_{2,k+1} \cup \mathsf{local}(s_k)$. However, since $D_2 \in \mathcal{D}_P$, by Definition 3.3 of distribution, we have that $\exists X_{2,\ell} \in D_2 : \mathsf{var}(\texttt{IF}\, e\, \texttt{THEN}\, \sigma_{\texttt{THEN}}\, \texttt{ELSE}\, \sigma_{\texttt{ELSE}}\, \texttt{END\_IF}) \subseteq X_{2,\ell} \cup \mathsf{local}(s_j)$. It follows that either $\ell = k$ or $\ell = k + 1$. Then, either $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{eval}[\nu_{1,\ell} \cup \mu_{1,j}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{eval}[\nu_{1,\ell} \cup \mu_{1,j}](e) = \mathsf{ff}$. In the case $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{tt}$, Rule (3.16) can be applied for process l from state q_2, leading to a state q_2' exactly like q_2 except for $\sigma_{2,j} = \sigma_{\texttt{THEN}};\sigma = \sigma_{1,j}$. In this case, we have that $\langle q_2', q_2'\rangle \in D$. In the case $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{tt}$, Rule (3.17) can be applied instead of Rule (3.16) and the same proof can be made with $\sigma_{\texttt{ELSE}}$ instead of $\sigma_{\texttt{THEN}}$.

Sequence Launch Rules If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.18) for some process i, we have that $\omega_{1,i} = \texttt{SEQ};\omega$ and that $\sigma_{1,j} = \texttt{LAUNCH}\, s_h; \sigma'_{2,j}$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ};\omega$, and that $\sigma_{2,j} = \sigma_{1,j} = \texttt{LAUNCH}\, s_h; \sigma'_{1,j}$. Therefore, Rule (3.18) can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\omega'_{2,i} = \texttt{LAUNCH}\, s_h; \omega_{2,i} = \texttt{LAUNCH}\, s_h; \omega_{1,i} = \omega'_{1,i}$ and $\sigma'_{2,j} = \sigma'_{1,j}$. Therefore, we can conclude that $\langle q_1', q_2'\rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\sigma_{2,j} = \sigma_{1,j} = \texttt{LAUNCH}\, s_h; \sigma'_{1,j}$ and that $\omega_{1,k} = \texttt{SEQ};\omega \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$, which implies, by Definition 3.9, that $\omega_{2,k} = \texttt{SEQ};\omega'$ and $\omega_{2,k+1} = \texttt{SEQ};\omega''$ with $\omega \prec \langle \omega', \omega''\rangle$. Therefore, Rule (3.18) can

be applied from q_2 for process k (it could also be applied for process $k+1$), leading to a state q_2' exactly like q_2 except for $\omega_{2,k}' = \texttt{LAUNCH}\, s_h; \omega_{2,k}$ and $\sigma_{2,j}' = \sigma_{1,j}'$. For the workloads, we know by Definition 3.9 of code distribution that $\texttt{LAUNCH}\, s_h \prec \langle \texttt{LAUNCH}\, s_h, \varepsilon \rangle$, and by definition of S, that $\omega_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$. It follows, by Lemma B.2, that $\omega_{1,k}' = \texttt{LAUNCH}\, s_h; \omega_{1,k} \prec \langle \texttt{LAUNCH}\, s_h; \omega_{2,k}, \omega_{2,k+1} \rangle = \langle \omega_{2,k}', \omega_{2,k+1}' \rangle$. Therefore, we can conclude that $\langle q_1', q_2' \rangle \in S$.

Sequence While Rules If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.19) or Rule (3.20) for some process i, we have that $\omega_{1,i} = \texttt{SEQ}; \omega$ and that $\sigma_{1,j} = \texttt{WHILE}\, e\, \texttt{DO}\, \sigma_b\, \texttt{END\_WHILE}; \sigma$. There are two cases to consider:

(*i*) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ}; \omega$, that $\nu_{2,i} = \nu_{1,i}$, that $\sigma_{2,j} = \sigma_{1,j} = \texttt{WHILE}\, e\, \texttt{DO}\, \sigma_b\, \texttt{END\_WHILE}; \sigma$ and that $\mu_{2,j} = \mu_{1,j}$. Since either Rule (3.19) or Rule (3.20) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,i} \cap \mathsf{local}(s_j) = X_{1,i} \cap \mathsf{local}(s_j)$. Then, either $\mathsf{eval}[\nu_{1,j} \cup \mu_{1,i}](e) = \mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{1,j} \cup \mu_{1,i}](e) = \mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{ff}$. In the case $\mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{tt}$, Rule (3.19) can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\sigma_{2,j}' = \sigma_b; \sigma_{2,j} = \sigma_b; \sigma_{1,j} = \sigma_{1,j}'$. In the case $\mathsf{eval}[\nu_{2,j} \cup \mu_{2,i}](e) = \mathsf{ff}$, Rule (3.20) can be applied from q_2, leading to a state q_2' exactly like q_2 except for $\sigma_{2,j}' = \sigma = \sigma_{1,j}'$. In both case, by definition of S, we have that $\langle q_1', q_2' \rangle \in S$.

(*ii*) If $i = k$, by definition of S, we have that $\sigma_{2,j} = \sigma_{1,j} = \texttt{WHILE}\, e\, \texttt{DO}\, \sigma_b\, \texttt{END\_WHILE}; \sigma$ and that $\omega_{1,k} = \texttt{SEQ}; \omega \prec \langle \omega_{2,k}, \omega_{2,k+1} \rangle$, which implies, by Definition 3.9 that $\omega_{2,k} = \texttt{SEQ}; \omega'$ and $\omega_{2,k+1} = \texttt{SEQ}; \omega''$ with $\omega \prec \langle \omega', \omega'' \rangle$. We also by definition of S that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1} \rangle$ and that $\mu_{1,j} = \mu_{2,j}$. Then, since Rule (3.19) of Rule (3.20) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,k} \cup \mathsf{local}(s_j) = X_{2,k} \cup X_{2,k+1} \cup \mathsf{local}(s_k)$. However, since $D_2 \in \mathcal{D}_P$, by Definition 3.3 of distribution, we have that $\exists X_{2,\ell} \in D_2 : \mathsf{var}(\texttt{WHILE}\, e\, \texttt{DO}\, \sigma_b\, \texttt{END\_WHILE}) \subseteq X_{2,\ell} \cup \mathsf{local}(s_j)$. It follows that either $\ell = k$ or $\ell = k+1$. Then, either $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{eval}[\nu_{1,\ell} \cup \mu_{1,j}](e) = \mathsf{tt}$, or $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{eval}[\nu_{1,\ell} \cup \mu_{1,j}](e) = \mathsf{ff}$. In the case $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{tt}$, Rule (3.19) can be applied for process ℓ from state q_2, leading to a state q_2' exactly like q_2 except for $\sigma_{2,j} = \sigma_b; \sigma_{2,j} = \sigma_b; \sigma_{1,j} = \sigma_{1,j}'$. In the case $\mathsf{eval}[\nu_{2,\ell} \cup \mu_{2,j}](e) = \mathsf{ff}$, Rule (3.20) can be applied for process ℓ from q_2, leading to a state q_2' exactly like q_2 except for $\sigma_{2,j}' = \sigma = \sigma_{1,j}'$. In both case, by definition of S, we have that $\langle q_1', q_2' \rangle \in S$.

Sequence Wait Rule If $q_1 \rightarrow_1 q_1'$ was derived using Rule (3.21) for some process i, we have that $\omega_{1,i} = \texttt{SEQ}; \omega$ and that $\sigma_{1,j} = \texttt{WAIT}(e); \sigma_{1,j}'$. There are two cases to consider:

(i) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ};\omega$, that $\nu_{2,i} = \nu_{1,i}$, that $\sigma_{2,j} = \sigma_{1,j} = \texttt{WAIT}(e);\sigma'_{1,j}$ and that $\mu_{2,j} = \mu_{1,j}$. Since Rule (3.21) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,i} \cup \mathsf{local}(s_j) = X_{2,i} \cup \mathsf{local}(s_j)$ and that $\mathsf{eval}[\mu_{1,j} \cup \nu_{1,i}](e) = \mathsf{eval}[\mu_{1,j} \cup \nu_{1,i}](e) = \mathsf{tt}$. Therefore, Rule (3.21) can be applied from q_2, leading to a state q'_2 exactly like q_2 except for $\sigma'_{2,j} = \sigma'_{1,j}$. Therefore, we can conclude that $\langle q'_1, q'_2\rangle \in S$.

(ii) If $i = k$, by definition of S, we have that $\sigma_{2,j} = \sigma_{1,j} = \texttt{WAIT}(e)\texttt{END\_WHILE};\sigma_{1,j}$ and that $\omega_{1,k} = \texttt{SEQ};\omega \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$, which implies, by Definition 3.9 that $\omega_{2,k} = \texttt{SEQ};\omega'$ and $\omega_{2,k+1} = \texttt{SEQ};\omega''$ with $\omega \prec \langle \omega', \omega''\rangle$. We also by definition of S that $\nu_{1,k} \prec \langle \nu_{2,k}, \nu_{2,k+1}\rangle$ and that $\mu_{1,j} = \mu_{2,j}$. Then, since Rule (3.21) was applied from q_1, it must be that $\mathsf{var}(e) \subseteq X_{1,k} \cup \mathsf{local}(s_j) = X_{2,k} \cup X_{2,k+1} \cup \mathsf{local}(s_k)$. However, since $D_2 \in \mathcal{D}_P$, by Definition 3.3 of distribution, we have that $\exists X_{2,\ell} \in D_2 : \mathsf{var}(\texttt{WAIT}(e)) \subseteq X_{2,\ell} \cup \mathsf{local}(s_j)$. It follows that either $\ell = k$ or $\ell = k + 1$. Moreover, we have that $\mathsf{eval}[\nu_{2,i} \cup \mu_{2,j}] = \mathsf{eval}[\nu_{1,i} \cup \mu_{1,j}] = \mathsf{tt}$. Therefore, Rule (3.21) can be applied for process ℓ, from q_2, leading to a state q'_2 exactly like q_2 except for $\sigma'_{2,j} = \sigma'_{1,j}$. Therefore, by definition of S, we can conclude that $\langle q'_1, q'_2\rangle \in S$.

End of Sequence Treatment Rule If $q_1 \rightarrow_1 q'_1$ was derived using Rule (3.22) for some process i, we have that $\omega_{1,i} = \texttt{SEQ};\omega'_{1,i}$. There are two cases to consider:

(i) If $i < k$, by definition of S, we have that $\omega_{2,i} = \omega_{1,i} = \texttt{SEQ};\omega'_{1,i}$ Therefore, Rule (3.22) we be applied from state q_2, leading to a state q'_2 exactly like q_2 except for $\omega'_{2,i} = \omega'_{1,i}$. By definition of S, we therefore have that $\langle q'_1, q'_2\rangle \in S$.

(ii) If $i = k$, by definition of S, we have that $\omega_{1,k} = \texttt{SEQ};\omega'_{1,k} \prec \langle \omega_{2,k}, \omega_{2,k+1}\rangle$ and by Definition 3.9 of code distribution that $\omega_{2,k} = \texttt{SEQ};\omega'_{2,k}$ and $\omega_{2,k+1} = \texttt{SEQ};\omega'_{2,k+1}$ with $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$. Therefore, we can first apply Rule (3.22) from q_2 for process k leading to a state q''_2. Note that Rule (3.22) does not change the valuations of the variables in V, which implies that $q_2 \overset{\sim}{\rightarrow}_{P,D_2} q''_2$. Then, from q''_2, since the workload of process $k + 1$ still begins with SEQ, we can again apply Rule (3.22), but this time for process $k+1$ (stuttering is needed for that) leading to a state q'_2 exactly like q_2 except for $\omega'_{2,k}$ and $\omega'_{2,k+1}$ defined above. Finally, since $\omega'_{1,k} \prec \langle \omega'_{2,k}, \omega'_{2,k+1}\rangle$, by definition of S, we can therefore conclude that $\langle q'_1, q'_2\rangle \in S$.

Appendix

C

dSL to Promela: an Example

In order to illustrate the translation of dSL programs into Promela, we introduce, in Section C.1, a small toy example and, in Section C.2, its corresponding Promela model.

C.1 The dSL Program

The dSL program is a simple program with four boolean variables, two events and two sequences, distributed on two execution sites. The source code follows.

```
 1 VAR
 2     x, y, z, t : BOOL;
 3 END_VAR
 4
 5 SITE Site1
 6     OUTPUT x : 1.1.1;
 7 END_SITE
 8
 9 SITE Site2
10     OUTPUT y : 1.1.1;
11 END_SITE
12
13 WHEN ~t THEN
14     y := TRUE;
15 END_WHEN
16
17 WHEN x THEN
18     t := TRUE;
19 END_WHEN
20
21 SEQUENCE foo()
22     x := TRUE;
23     z := TRUE;
24     y := FALSE;
25     z := FALSE;
26 END_SEQUENCE
27
28 SEQUENCE main()
29     LAUNCH foo();
30 END_SEQUENCE
```

C.2 The Corresponding Promela Code

```
1  #define N_SITES             2
2  #define GAMMA               2
3
4  #define SITE_Site1          1
5  #define SITE_Site2          2
6
7  #define SEQUENCE_foo       15
8  #define SEQUENCE_main      16
9
10 /****************** Global Variables ***************/
11
12 chan phi_Site1[N_SITES] = [GAMMA] of { byte, byte };
13 chan phi_Site2[N_SITES] = [GAMMA] of { byte, byte };
14
15 bool x, y, z, t, tilde_t[N_SITES];
16 int foo_site, foo_pc, main_site, main_pc;
17 bool event_13_line_13_old_cond, event_14_line_17_old_cond;
18
19 /******************** IO Inlines *******************/
20
21 /* change this for environment dependent behaviour! */
22
23 inline input() { skip; }
24 inline output() { skip; }
25
26 /**************** Events & Sequences ****************/
27
28 inline event_13_line_13(pc) {
29     pc = 0;
30     cond = tilde_t[local_site-1];
31     if
32     :: (!event_13_line_13_old_cond && cond) -> {
33            do
34            :: (pc == 0) -> {
35                   y = 1;
36                   break;
37               }
38            od;
39        }
40     :: else -> skip;
41     fi;
42     event_13_line_13_old_cond = cond;
43 }
```

```
44
45 inline event_14_line_17(pc) {
46     pc = 0;
47     cond = x;
48     if
49     :: (!event_14_line_17_old_cond && cond) -> {
50            do
51            :: (pc == 0) -> {
52                   t = 1; broadcast(8, t);
53                   break;
54               }
55            od;
56        }
57     :: else -> skip;
58     fi;
59     event_14_line_17_old_cond = cond;
60 }
61
62
63 inline sequence_foo(site, pc) {
64     do
65     :: (pc == -1) -> skip;
66     :: (pc == 0) -> {
67            x = 1; event_14_line_17(local_pc);
68            z = 1;
69            pc = 1;
70            site = SITE_Site2;
71            break;
72        }
73     :: (pc == 1) -> {
74            y = 0;
75            pc = 2;
76            site = SITE_Site1;
77            break;
78        }
79     :: (pc == 2) -> {
80            z = 0;
81            pc = -1;
82            break;
83        }
84     od;
85 }
86
87
```

```
88 inline sequence_main(site, pc) {
89     do
90     :: (pc == -1) -> skip;
91     :: (pc == 0) -> {
92           if
93           :: (foo_pc == -1) -> foo_pc = 0; foo_site = SITE_Site1;
94           :: else -> skip;
95           fi;
96           pc = -1;
97           break;
98        }
99     od;
100 }
101
102
103 /****************** Internal Inlines *****************/
104
105 inline treat_messages(channel) {
106     n_messages[0] = len(channel[0]);
107     n_messages[1] = len(channel[1]);
108     do
109     :: if
110        :: n_messages[0] -> n_messages[0]--; channel[0] ? var, val;
111        :: n_messages[1] -> n_messages[1]--; channel[1] ? var, val;
112        :: else -> break;
113        fi;
114        if
115        :: (var == 8) -> {
116              tilde_t[local_site-1] = val;
117              if
118              :: (local_site == SITE_Site2) -> event_13_line_13(local_pc);
119              :: else -> skip;
120              fi
121           }
122        fi;
123     :: break;
124     od;
125 }
126
127 inline broadcast(var, val) {
128     phi_Site1[local_site] ! var, val;
129     phi_Site2[local_site] ! var, val;
130 }
131
```

```
132 inline treat_sequences() {
133     do
134     :: (foo_site == local_site) -> sequence_foo(foo_site, foo_pc);
135     :: (main_site == local_site) -> sequence_main(main_site, main_pc);
136     :: break;
137     od;
138 }
139
140 /********************** Sites **********************/
141
142 proctype Site1() {
143     int n_messages[N_SITES], var, val;
144     int local_site = SITE_Site1, local_pc;
145     bool cond;
146     do
147     :: input(); event_14_line_17(local_pc);
148        treat_messages(phi_Site1);
149        treat_sequences();
150        output();
151     od;
152 }
153
154 proctype Site2() {
155     int n_messages[N_SITES], var, val;
156     int local_site = SITE_Site2, local_pc;
157     bool cond;
158     do
159     :: input(); event_13_line_13(local_pc);
160        treat_messages(phi_Site2);
161        treat_sequences();
162        output();
163     od;
164 }
165
166 init {
167     foo_pc = -1;
168     main_pc = -1;
169     main_site = SITE_Site1;
170     main_pc = 0;
171     run Site1();
172     run Site2();
173 }
```

Appendix

D

The Canal Locks Controllers

```
1  (* Types *)
2  CLASS GATE
3    motorCommand, motorDirection        : BOOL;
4    opened, closed, motorOrderGiven     : BOOL;
5    buttonOpen, buttonClose             : BOOL;
6  END_CLASS
7
8  CLASS LOCK
9    valveCommand, valveDirection        : BOOL;
10   levelUp, levelDown, valveOrderGiven : BOOL;
11   bottomGate, topGate                 : GATE;
12   buttonFill, buttonEmpty             : BOOL;
13 END_CLASS
14
15 (* Global variables *)
16 VAR
17   lock1, lock2                        : LOCK;
18   notAllowed                          : BOOL;
19 END_VAR
20
21 (* Site *)
22 SITE lowerLock
23   INPUT lock1.bottomGate.opened          : 1.0.1;
24   INPUT lock1.bottomGate.closed          : 1.0.2;
25   INPUT lock1.topGate.opened             : 1.0.3;
26   INPUT lock1.topGate.closed             : 1.0.4;
27   INPUT lock1.levelDown                  : 1.0.5;
28   INPUT lock1.levelUp                    : 1.0.6;
29   OUTPUT lock1.bottomGate.motorCommand   : 2.0.1;
30   OUTPUT lock1.bottomGate.motorDirection : 2.0.2;
31   OUTPUT lock1.topGate.motorCommand      : 2.0.3;
```

```
32   OUTPUT lock1.topGate.motorDirection    : 2.0.4;
33   OUTPUT lock1.valveCommand              : 2.0.5;
34   OUTPUT lock1.valveDirection            : 2.0.6;
35 END_SITE
36
37 SITE upperLock
38   INPUT lock2.bottomGate.opened          : 1.0.1;
39   INPUT lock2.bottomGate.closed          : 1.0.2;
40   INPUT lock2.topGate.opened             : 1.0.3;
41   INPUT lock2.topGate.closed             : 1.0.4;
42   INPUT lock2.levelDown                  : 1.0.5;
43   INPUT lock2.levelUp                    : 1.0.6;
44   OUTPUT lock2.bottomGate.motorCommand   : 2.0.1;
45   OUTPUT lock2.bottomGate.motorDirection : 2.0.2;
46   OUTPUT lock2.topGate.motorCommand      : 2.0.3;
47   OUTPUT lock2.topGate.motorDirection    : 2.0.4;
48   OUTPUT lock2.valveCommand              : 2.0.5;
49   OUTPUT lock2.valveDirection            : 2.0.6;
50 END_SITE
51
52 SITE controlPanel
53   INPUT lock1.bottomGate.buttonOpen      : 1.0.1;
54   INPUT lock1.bottomGate.buttonClose     : 1.0.2;
55   INPUT lock1.topGate.buttonOpen         : 1.0.3;
56   INPUT lock1.topGate.buttonClose        : 1.0.4;
57   INPUT lock1.buttonFill                 : 1.0.5;
58   INPUT lock1.buttonEmpty                : 1.0.6;
59   INPUT lock2.bottomGate.buttonOpen      : 1.0.7;
60   INPUT lock2.bottomGate.buttonClose     : 1.0.8;
61   INPUT lock2.topGate.buttonOpen         : 1.0.9;
62   INPUT lock2.topGate.buttonClose        : 1.0.10;
63   INPUT lock2.buttonFill                 : 1.0.11;
64   INPUT lock2.buttonEmpty                : 1.0.12;
65   OUTPUT notAllowed                      : 2.0.1;
66 END_SITE
67
68 (* Methods *)
69 METHOD GATE::move(direction : BOOL)
70   self.motorDirection := direction;
71   self.motorCommand := TRUE;
72 END_METHOD
73
74 METHOD GATE::resetOrderGiven()
75   self.motorOrderGiven := FALSE;
```

```
76   notAllowed := FALSE;
77 END_METHOD
78
79 METHOD LOCK::waterMove(direction : BOOL)
80   IF NOT self.levelDown THEN
81     self.valveCommand := TRUE;
82     self.valveDirection := direction;
83   END_IF;
84 END_METHOD
85
86 METHOD LOCK::resetValveOrderGiven()
87   self.valveOrderGiven := FALSE;
88   notAllowed := FALSE;
89 END_METHOD
90
91 (* Events *)
92 WHEN IN GATE self.closed OR self.opened THEN
93   self.motorCommand := FALSE;
94   LAUNCH self<-resetOrderGiven();
95 END_WHEN
96
97 WHEN IN LOCK self.levelUp OR self.levelDown THEN
98   self.valveCommand := FALSE;
99   LAUNCH self<-resetValveOrderGiven();
100 END_WHEN
101
102 WHEN lock1.bottomGate.buttonOpen THEN
103   IF (~lock1.topGate.closed) AND (NOT lock1.topGate.motorOrderGiven) AND
104      (~lock1.levelDown) AND (NOT lock1.valveOrderGiven)
105   THEN
106     notAllowed := FALSE;
107     lock1.bottomGate.motorOrderGiven := TRUE;
108     LAUNCH lock1.bottomGate<-move(TRUE);
109   ELSE
110     notAllowed := TRUE;
111   END_IF;
112 END_WHEN
113
114 WHEN lock1.topGate.buttonOpen THEN
115   IF (~lock1.bottomGate.closed) AND (NOT lock1.bottomGate.motorOrderGiven) AND
116      (~lock2.bottomGate.closed) AND (NOT lock2.bottomGate.motorOrderGiven) AND
117      (~lock1.levelUp) AND (NOT lock1.valveOrderGiven)
118   THEN
119     notAllowed := FALSE;
```

```
120     lock1.topGate.motorOrderGiven := TRUE;
121     LAUNCH lock1.topGate<-move(TRUE);
122   ELSE
123     notAllowed := TRUE;
124   END_IF;
125 END_WHEN
126
127 WHEN lock2.bottomGate.buttonOpen THEN
128   IF (~lock1.topGate.closed) AND (NOT lock1.topGate.motorOrderGiven) AND
129      (~lock2.topGate.closed) AND (NOT lock2.topGate.motorOrderGiven) AND
130      (~lock2.levelDown) AND (NOT lock2.valveOrderGiven)
131   THEN
132     notAllowed := FALSE;
133     lock2.bottomGate.motorOrderGiven := TRUE;
134     LAUNCH lock2.bottomGate<-move(TRUE);
135   ELSE
136     notAllowed := TRUE;
137   END_IF;
138 END_WHEN
139
140 WHEN lock2.topGate.buttonOpen THEN
141   IF (~lock2.bottomGate.closed) AND (NOT lock2.bottomGate.motorOrderGiven) AND
142      (~lock2.levelUp) AND (NOT lock2.valveOrderGiven)
143   THEN
144     notAllowed := FALSE;
145     lock2.topGate.motorOrderGiven := TRUE;
146     LAUNCH lock2.topGate<-move(TRUE);
147   ELSE
148     notAllowed := TRUE;
149   END_IF;
150 END_WHEN
151
152 WHEN IN GATE self.buttonClose THEN
153   LAUNCH self<-move(FALSE);
154 END_WHEN
155
156 WHEN IN LOCK self.buttonFill THEN
157   IF (~self.bottomGate.closed) AND (NOT self.bottomGate.motorOrderGiven) AND
158      (~self.topGate.closed) AND (NOT self.topGate.motorOrderGiven)
159   THEN
160     notAllowed := FALSE;
161     self.valveOrderGiven := TRUE;
162     LAUNCH self<-waterMove(TRUE);
163   ELSE
```

```
164     notAllowed := TRUE;
165   END_IF;
166 END_WHEN
167
168 WHEN IN LOCK self.buttonEmpty THEN
169   IF (~self.bottomGate.closed) AND (NOT self.bottomGate.motorOrderGiven) AND
170      (~self.topGate.closed) AND (NOT self.topGate.motorOrderGiven)
171   THEN
172     notAllowed := FALSE;
173     self.valveOrderGiven := TRUE;
174     LAUNCH self<-waterMove(FALSE);
175   ELSE
176     notAllowed := TRUE;
177   END_IF;
178 END_WHEN
179
180 (* Sequences *)
181 SEQUENCE main()
182   notAllowed := FALSE;
183 END_SEQUENCE
```

List of Definitions

1.1 Syntax of Propositional Boolean Logic 18
1.2 Semantics of PBL 19
1.3 Syntax of Quantified Boolean Logic 19
1.4 Semantics of QBL 19
1.5 Finite Automaton 21
1.6 Transition System 22
1.7 Kripke Structure 23
1.8 Simulation Relation 24
1.9 Stuttering Simulation Relation 25
1.10 Syntax of Linear Temporal Logic 27
1.11 Semantics of LTL 27
1.12 Syntax of Computational Tree Logic 29
1.13 Semantics of CTL 29

3.1 dSL Program 52
3.2 Well-formed dSL Program 55
3.3 Distribution of a dSL Program 55
3.4 Coarser/Finer Distribution 56
3.5 Local State of a Process 60
3.6 Local State of a Sequence 61
3.7 Global State of a dSL Program 61
3.8 Semantics of a dSL Program 69
3.9 Code Distribution 71
3.10 Valuation Distribution 71
3.11 Channel Distribution 71
3.12 Communicating Finite State Machine 75
3.13 Semantics of Communicating Finite State Machine 75

3.14 Triggering Depth 81

4.1 Vector Clocks 89
4.2 Happened-Before Relation [Lamport, 1978] 90
4.3 Partial Order Trace 94
4.4 Semantics of Partial Order Traces 95
4.5 Predicate Detection Problem 98
4.6 Semantics of LTL over Finite Sequences 101
4.7 LTL Trace Checking Problem 101
4.8 Semantics of CTL over Partially Ordered Traces 105
4.9 CTL Trace Checking Problem 105

5.1 Local Predicate [Charron-Bost et al., 1995] 114
5.2 Disjunctive Predicate 115
5.3 Stable Predicate [Chandy and Lamport, 1985] 117
5.4 Observer-Independent Predicate [Charron-Bost et al., 1995] 118
5.5 Conjunctive Predicate 120
5.6 Linear Predicate [Chase and Garg, 1995; Garg et al., 2003] 123
5.7 Regular Predicate [Garg and Mittal, 2001b] 124
5.8 Computation Graph 125
5.9 Slice [Garg and Mittal, 2005] 127
5.10 Meet-Closed Predicate 133

6.1 Monitor 140
6.2 Composition 145
6.3 Determined Monitor 149
6.4 Monitor Driven Composition 155
6.5 Covering Operator 160

7.1 Regular Temporal Predicates 168
7.2 Syntax of RCTL [Sen and Garg, 2003] 169
7.3 Tuple Representation of Cuts 177
7.4 Interval Sharing Tree 191

List of Theorems

1.1 Lattice of Order Ideals [Priestley and Davey, 2002] 15
1.2 Representation of Finite Distributive Lattice [Brikhoff, 1940] 16
1.3 Fixed Point Characterisation [Tarski, 1955; Knaster, 1928] 16
1.4 Complexity of PBL-SAT [Cook, 1971] 19
1.5 Complexity of QBL-SAT [Stockmeyer and Meyer, 1973] 20
1.6 Simulation and Traces 24
1.7 Stuttering Simulation and Traces 26
1.8 Complexity of LTL-MC [Sistla and Clarke, 1985] 28
1.9 LTL-X is Stuttering-Closed [Lamport, 1983] 28
1.10 Stuttering Simulation Preserves LTL-X 29
1.11 Complexity of CTL-MC [Clarke et al., 1983] 30
1.12 Fixed Point Characterization of CTL [Emerson and Clarke, 1980] 30

3.1 Partition of Events 56
3.2 Lattice of Distributions 57
3.3 Stuttering Simulation of the Semantics 73
3.4 dSL-LTL-MC: A Sufficient Condition for LTL-X Properties 74
3.5 Undecidability of CFSM-REACH [Brand and Zafiropulo, 1983] 76
3.6 Undecidability of Model Checking for dSL programs 78
3.7 Stuttering Simulation of the Bounded Semantics 83

4.1 Lattice of Cuts 97
4.2 Complexity of PRED [Chase and Garg, 1995] 99
4.3 Complexity of PRED for Total Order Traces 99
4.4 Complexity of LTL-TC for Total Order Traces 102
4.5 Complexity of LTL-TC 104
4.6 Complexity of CTL-TC 108

4.7 Complexity of CTL-TC for Total Order Traces 108

5.1 The Lattice of Cuts is Distributive [Garg and Mittal, 2005] 125
5.2 Lattice of Consistent Cuts [Garg and Mittal, 2005] 127
5.3 Uniqueness of Consistent Cuts in Slices [Garg and Mittal, 2005] 127
5.4 Slicing w.r.t Regular Predicates [Garg and Mittal, 2005] 130
5.5 Linear and Meet-Closed Coïncide [Chase and Garg, 1998] 133
5.6 Litterals are Observer-Independent 134
5.7 Litterals are Regular 136
5.8 Complexity of PRED for PBL Formulae in DNF 137

6.1 Correctness of the composition 146
6.2 Necessary and Sufficient Conditions for Valuation Equivalence 151
6.3 Correctness of Algorithm 6.3 153
6.4 Existence of a Determined Monitor 154
6.5 Correctness of the Monitor Driven Composition 159
6.6 Correctness of Algorithm 6.4 162

7.1 Regular-Preserving CTL Modalities [Sen and Garg, 2003] 168
7.2 Correctness of Algorithm 7.2 180
7.3 Correctness of Algorithm 7.3 184
7.4 Tuple Characterization of pre(X) 187
7.5 Tuple Characterization of EF 188

List of Algorithms

1.1 Fixed point computation in a complete lattice 17

4.1 Mattern's vector clock algorithm for message passing program 90
4.2 Sen's vector clock algorithm for multithreaded program 92
4.3 Satisfaction of LTL formulae over finite sequences [Markey, 2003] 103

5.1 Detection of arbitrary predicates 113
5.2 Detection of local predicates 115
5.3 Detection of disjunctive predicates 116
5.4 Detection of stable predicate 117
5.5 Detection of observer-independent predicate 119
5.6 Detection of conjunctive predicate 122
5.7 Detection of linear predicate 124
5.8 Computation of a slice of a po-trace w.r.t a predicate 128
5.9 Detection of regular predicates 131
5.10 Detection of arbitrary predicates using slicing 132
5.11 Detection of DNF predicates 137

6.1 Construction of monitors for LTL formulae 142
6.1 Construction of monitors for LTL formulae (cont'd) 143
6.2 Explicit LTL trace checking algorithm 147
6.3 Construction of determined monitors for LTL formulae 152
6.4 Symbolic LTL trace checking algorithm 160

7.1 RCTL trace checking algorithm 175
7.2 Computation of the set of tuples satisfying $\top$ 178
7.3 Computation of the set of tuples satisfying a proposition 181

7.4 Computation of tuple(pre(X)) and tuple($\widetilde{\text{pre}}(X)$) 187
7.5 CTL trace checking algorithm 189
7.5 CTL trace checking algorithm (cont'd) 190

List of Figures

1 The model checking process 4
2 The testing process 6

1.1 Examples of posets 15
1.2 Illustration of Birkhoff's representation theorem 16
1.3 Example of finite automata 21
1.4 Examples of Kripke Structures 24
1.5 Example of (stuttering) simulation 26
1.6 Example of KSs satisfying, or not, some CTL formulae 31

2.1 Overview of a SL system 34
2.2 Overview of a dSL system 35
2.3 A boiler control program 37
2.4 Class declaration 38
2.5 Global variable declaration 39
2.6 Site declaration 40
2.7 Method declaration and call 41
2.8 Event declaration 42
2.9 Example of undistributable program 42
2.10 Sequence declaration and call 44
2.11 Localising atomic code 46
2.12 Localising sequential code 48

3.1 Variables and instructions appearing in a dSL construct 54
3.2 A bogus dSL program 58
3.3 The lattice of distributions for the program of Figure 3.2 59
3.4 Valuations extended to dSL expressions 61
3.5 Events used to simulate a CFSM 79
3.6 A dSL program with an infinite triggering loop 80

3.7 A dSL program with possibly unbounded channels channels 82

4.1 Centralised vs. Distributed system observation 88
4.2 Execution of a message passing program with vector clock mapping 91
4.3 Execution of a shared variable program with vector clock mapping 93
4.4 Example of po-trace 95
4.5 Semantics of the po-trace of Figure 4.4 96

5.1 A po-trace where $p \wedge q$ is local to P_2 114
5.2 The po-trace T of Example 5.6 120
5.3 Partial order of the po-trace of Figure 4.4(a) extended to local states 121
5.4 Example of computation graph 126
5.5 Algorithm 5.8 at work 129
5.6 A hierarchy of predicates 138

6.1 Construction of a monitor for F$(p \vee q)$ - graph expansion 144
6.2 Construction of a monitor for F $(p \vee q)$ - reduction 145
6.3 $V \not\equiv_s V'$ 149
6.4 Construction of a determined monitor for F$(p \wedge q)$ 153
6.5 Monitor sensitivity 155
6.6 Monitor driven composition 156
6.7 Experimental results ($\uparrow\uparrow$ indicates $>$ 1GB) 163

7.1 Counter-examples for the regularity of regular temporal modalities 169
7.2 Computation slicing for the EF modality 171
7.3 Computation slicing for the AG modality 173
7.4 Computation slicing for the EG modality 174
7.5 Computation of the set of tuples satisfying $\top$ 179
7.6 Computation of the set tuples satisfying a proposition 182
7.7 Computation of the IST for $[\![\top]\!]^T_{\mathbb{N}^k}$ on the po-trace of Figure 4.4 192
7.8 Experimental results ($\uparrow\uparrow$ indicates $>$ 10 min.) 193

8.1 A canal lock system 196
8.2 Excerpt from the dSL source code of the canal lock controller 197
8.2 Flip-Flop behaviour 199
8.3 Results of the verification of the canal lock controller 200

8.4 Error in the canal locks controller 201
8.5 dSL code dealing with the opening of the bottom gate of the upper lock 202
8.6 Overview of the C simulator 203
8.7 Results of the testing of the canal lock controller 204
8.8 Detailed analysis of one trace 205

Bibliography

[Aaronson and Kuperberg, 2005] Aaronson, S. and Kuperberg, G. (2005). Complexity Zoo. `http://qwiki.caltech.edu/wiki/Complexity_Zoo`.

[Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer.

[Abdulla et al., 1996] Abdulla, P. A., Cerans, K., Jonsson, B., and Tsay, Y.-K. (1996). General Decidability Theorems for Infinite-State Systems. In *Proceedings of the 11th Annual Symposium on Logics in Computer Science (LICS'96), New Brunswick (USA)*, pages 313–321. IEEE Computer Society.

[Abrial, 1996] Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

[Aho et al., 1998] Aho, A. V., Sethi, R., and Ullman, J. D. (1998). *Compilers: Principles, Techniques and Tools*. Addison Wesley.

[Alur and Dill, 1990] Alur, R. and Dill, D. L. (1990). Automata For Modeling Real-Time Systems. In *Proceeding of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90), Warwick University (England)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag.

[Alur et al., 1995] Alur, R., Peled, D., and Penczek, W. (1995). Model-Checking of Causality Properties. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'05), San Diego (USA)*, pages 90–100. IEEE Computer Society.

[Ammirati et al., 2002] Ammirati, P., Delzanno, G., Ganty, P., Geeraerts, G., Raskin, J.-F., and Van Begin, L. (2002). Babylon: an Integrated Toolkit for the Specification and Verification of Parameterized Systems. In *Proceedings of the 2nd Workshop on Specification, Analisys and Validation for Emerging Technologies (SAVE'02), Copenhagen (Denmark)*.

[Aubry, 1997] Aubry, P. (1997). *Mises en Oeuvre Distribuées de Programmes Synchrones*. PhD thesis, Institut de Formation Supérieure en Informatique et Communication.

[Augusto et al., 2003] Augusto, J. C., Howard, Y., Gravell, A. M., Ferreira, C., Gruner, S., and Leuschel, M. (2003). Model-based approaches for validating business critical systems. In *Proceedings of the 11thInternational Workshop on Software Technology and Engineering Practice (STEP'03), Amsterdam (The Netherlands*, pages 225–233. IEEE Computer Society Press.

[Ball and Rajamani, 2002] Ball, T. and Rajamani, S. K. (2002). Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical report, Microsoft Research.

[Bauer et al., 2006] Bauer, A., Leucker, M., and Schallhart, C. (2006). Monitoring of Real-Time Properties. In *Proceedings of the 26thInternational Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06), Kolkata (India)*, volume 4337 of *Lectur Notes in Computer Science*, pages 260–272. Springer-Verlag.

[Bengtsson et al., 1995] Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). Uppaal : a Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of the 3rdWorkshop on Verification and Control on Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag.

[Benveniste and Berry, 1991] Benveniste, A. and Berry, G. (1991). The Synchronous Approach to Reactive and Real-Time Systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1282.

[Berry, 1989] Berry, G. (1989). Real Time Programming: Special Purpose or General Purpose Languages. *Information Processing Letter*, pages 11–18.

[Berry and Gonthier, 1992] Berry, G. and Gonthier, G. (1992). The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152.

[Biere et al., 2003] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, 58:118–149.

[Bouajjani et al., 1997] Bouajjani, A., Esparza, J., and Maler, O. (1997). Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceeding of the 8thInternational Conference on Concurrency Theory (CONCUR'97), Warsaw (Poland)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag.

[Boudet and Comon, 1996] Boudet, A. and Comon, H. (1996). Diophantine Equations, Presburger Arithmetic and Finite Automata. In *Proceedings of the 21stInternational Colloquium on Trees in Algebra and Programmings (CAAP'96), Linköping (Sweden)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag.

[Bramer, 2005] Bramer, M. A. (2005). *Logic Programming with PROLOG*. Springer.

[Brand and Zafiropulo, 1983] Brand, D. and Zafiropulo, P. (1983). On Communicating Finite-State Machines. *Jounrla of the ACM*, 30(2):323–342.

[Brikhoff, 1940] Brikhoff, G. D. (1940). *Lattice Theory*. American Mathematical Society.

[Brinksma and Langerak, 1995] Brinksma, E. and Langerak, R. (1995). Functionality Decomposition by Compositional Correctness Preserving Transformation. *South African Computer Journal*, 13:2–13.

[Browne et al., 1988] Browne, M. C., Clarke, E. M., and Grumberg, O. (1988). Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318.

[Büchi, 1960] Büchi, J. R. (1960). On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science, Stanford (USA)*, pages 1–11.

[Caspi et al., 1987] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. (1987). LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Proceedings of the 14thACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87), Munich (West Germany)*, pages 178–188.

[Castellani et al., 1999] Castellani, I., Mukund, M., and Thiagarajan, P. S. (1999). Synthesizing Distributed Transition Systems from Global Specification. In *Proceedings of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95), Bangalore (India)*, volume 1738 of *Lecture Notes In Computer Science*, pages 219–231. Springer-Verlag.

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Science (TOCS)*, 3(1):63–75.

[Chandy and Misra, 1988] Chandy, K. M. and Misra, J. (1988). *Parallel Program Design: a Foundation*. Addison-Wesley.

[Charron-Bost et al., 1995] Charron-Bost, B., Delporte-Gallet, C., and Fauconnier, H. (1995). Local and Temporal Predicates in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179.

[Chase and Garg, 1998] Chase, C. G. and Garg, V. K. (1998). Detection of Global Predicates: Techniques and Their Limitations. *Distributed Computing*, 11(4):191–201.

[Chase and Garg, 1995] Chase, C. M. and Garg, V. K. (1995). Efficient detection of restricted classes of global predicates. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95), Le Mont-Saint-Michal (France)*, volume 972 of *Lecture Notes in Computer Science*, pages 303–317. Springer.

[Chatterjee et al., 2003] Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T. A., and Palsberg, J. (2003). Stack Size Analysis for Interrupt-Driven Programs. In *Proceedings of the 10th International Symposium on Static Analysis (SAS'03), San Diego (USA)*, volume 2694 of *Lecture Notes in Computer Science*, pages 109–126. Springer.

[Church, 1936] Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345–363.

[Cimatti et al., 2002] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV 2: An Open-Source Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02), Copenhagen (Denmark)*, volume 20404 of *Lecture Notes in Computer Science*, pages 359–364. Springer.

[Cimatti et al., 1999] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NUSMV: A New Symbolic Model Verifier. In *Proceedings of the 11thInternational Conference on Computer Aided Verification (CAV'99), Trento (Italy)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer.

[Clarke and Emerson, 1981] Clarke, E. M. and Emerson, A. E. (1981). Design and Systhesis of Synchronization Skeleton Using Branching Time Temporal Logic. In *Proceedings of the 3rdWorkshop on Logic For Programs, New York (USA)*, volume 407 of *Lecture Notes in Computer Science*, pages 55–71. Springer-Verlag.

[Clarke et al., 1983] Clarke, E. M., Emerson, A. E., and Sistla, P. A. (1983). Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications : A Practical Approach. In *Proceedings of the 10thACM Symposion of Principle of Programming Languages (POPL'83), Austin (USA)*, pages 117–126. ACM Press.

[Clarke et al., 1999] Clarke, E. M., Grunberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[Cook, 1971] Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. In *Proceeding of the 3rdAnnual ACM Symposium on Theory of Computing, Shaker Heights (USA)*, pages 151–158. ACM Press.

[Dahlhaus et al., 1994] Dahlhaus, E., Johnson, D. S., Papadimitriou, C. H., Seymour, P. D., and Yannakakis, M. (1994). The Complexity of Multiterminal Cuts. *SIAM Journal of Computing*, 23(4):864–894.

[De Wachter, 2005] De Wachter, B. (2005). dSL*, a Language and Environment for the Design of Distributed Industrial Controllers*. PhD thesis, Université Libre de Bruxelles.

[De Wachter et al., 2005] De Wachter, B., Genon, A., and Massart, T. (2005). From Static Code Distribution to More Shrinkage for the Multiterminal Cut. In *Proceedings of the 4thInternational Workshop on Experimental and Efficient Algorithms (WEA'05), Santorini Island (Greece)*, volume 3503 of *Lecture Notes in Computer Science*, pages 177–188. Springer.

[De Wachter et al., 2005] De Wachter, B., Genon, A., Massart, T., and Meuter, C. (2005). The Formal Design of Distributed Controllers with dSL and Spin. *Formal Aspect of Computing*, 17(2):177–200.

[De Wachter et al., 2003a] De Wachter, B., Massart, T., and Meuter, C. (2003a). An Experiment on Synthesis and Verification of an Industrial Process Control in the dSL Environnment. In *Proceeding of the 3rd Internaltional Workshop on Automated Verification of Critical Systems (AVoCS'03), Southampton (UK).*

[De Wachter et al., 2003b] De Wachter, B., Massart, T., and Meuter, C. (2003b). dSL: An Environment with Automatic Code Distribution for Industrial Control Systems. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS'03), La Martinique (France)*, volume 3144 of *Lecture Notes in Computer Sciences*, pages 132–145. Springer-Verlag.

[Detlefs et al., 2003] Detlefs, D., Nelson, G., and Saxe, J. B. (2003). Simplify: A Theorem Prover for Program Checking. Technical report, System Research Center, HP Laboratories Palo Alto.

[Diekert and Gastin, 2002] Diekert, V. and Gastin, P. (2002). LTL Is Expressively Complete for Mazurkiewicz Traces. *Journal of Computer and System Sciences*, 64(2):396–418.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded Commands, Non-Determinacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457.

[Dijkstra and Scholten, 1990] Dijkstra, E. W. and Scholten, C. S. (1990). *Predicate Calculus and Program Semantics*. Springer-Verlag.

[Emerson and Clarke, 1980] Emerson, A. E. and Clarke, E. M. (1980). Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming, Noordweijkerhout (The Netherland)*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Spinger-Verlag.

[Eskicioglu, 1990] Eskicioglu, M. R. (1990). Design Issues of Process Migration Facilities in Distributed Systems. *IEEE Computer Society Technical Committe on Operating Systems and Application Environments Newsletter*, 4:3–13.

[Esparza and Heljanko, 2001] Esparza, J. and Heljanko, K. (2001). Implementing ltl model checking with net unfoldings. In *Proceedings of the 8th International SPIN Workshop (SPIN'01), Toronto (Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag.

[Fidge, 1991] Fidge, C. (1991). Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33.

[Fleury, 2002] Fleury, E. (2002). *Automates Temporisés avec MiseS à Jour*. PhD thesis, École Normale Supérieur de Cachan.

[Ganty, 2002] Ganty, P. (2002). Algorithmes et structures de données efficaces pour la manipulation de contraintes sur les intervalles. Master's thesis, Université Libre de Bruxelles.

[Ganty et al., 2006] Ganty, P., Meuter, C., Van Begin, L., Kalyon, G., Raskin, J.-F., and Delzanno, G. (2006). Symbolic Data Structure for sets of k-uples of Integers. Technical Report 570, Université Libre de Bruxelles.

[Garg, 1992] Garg, V. K. (1992). Some Optimal Algorithms for Decomposed Partially Ordered Sets. *Information Processing Letter*, 44:39–43.

[Garg and Mittal, 2001a] Garg, V. K. and Mittal, N. (2001a). Computation Slicing: Techniques and Theory. In *Proceedings of the 15thInternaltional Conference on Distributed Computing (DISC'01), Lisbon (Portugal)*, volume 2180 of *Lecture Notes in Computer Science*, pages 78–92. Springer.

[Garg and Mittal, 2001b] Garg, V. K. and Mittal, N. (2001b). On Slicing a Distributed Computation. In *Proceedings of the 21stInternational Conference on Distributed Computing Systems (ICDCS'01), Phoenix (USA)*, pages 322–329.

[Garg and Mittal, 2005] Garg, V. K. and Mittal, N. (2005). Techniques and Applications of Computation Slicing. *Distributed Computing*, 17(3):251–277.

[Garg et al., 2003] Garg, V. K., Mittal, N., and Sen, A. (2003). Applications of Lattice Theory to Distributed Computing. *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT) News*, 34(3):40–61.

[Garg and Waldecker, 1994] Garg, V. K. and Waldecker, B. (1994). Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307.

[Genon, 2004] Genon, A. (2004). *On the Verification of* dSL*, a Language to Design Distributed Industrial Control Systems*. DEA Thesis, Université Libre de Bruxelles.

[Genon et al., 2006] Genon, A., Massart, T., and Meuter, C. (2006). Monitoring Distributed Controllers: When an Efficient LTL Algorithm on Sequences Is Needed to

Model-Check Traces. In *Proceedings of the 14thInternational Symposium on Formal Methods (FM'06), Hamilton (Canada)*, volume 4085 of *Lecture Notes in Computer Science*, pages 557–572. Springer-Verlag.

[Gerth et al., 1995] Gerth, R., Peled, D., Vardi, M. Y., and Wolpe, P. (1995). Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15thIFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV'95), Warsaw (Poland)*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall.

[Giannakopoulou and Havelund, 2001] Giannakopoulou, D. and Havelund, K. (2001). Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16thInternational Conference on Automated Software Engineering (ASE'01), San Diego (USA)*, pages 412–416. IEEE Computer Society.

[Girault, 1994] Girault, A. (1994). *Sur la Répartition de Programmes Synchrones*. PhD thesis, Institut National Polytechnique de Grenoble.

[Godefroid, 1996] Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Holzmann, 1997] Holzmann, G. J. (1997). The Model Checker Spin. *IEEE Transactions on Software Enfineering*, 23(5):279–295.

[Holzmann, 2004] Holzmann, G. J. (2004). *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional.

[Hopcroft et al., 2000] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2ndedition.

[Jard et al., 1994] Jard, C., Jéron, T., Jourdan, G.-V., and Rampon, J.-X. (1994). A General Approach to Trace-Checking in Distributed Computing Systems. In *Proceedings of the 14thInternational Conference on Distributed Computing Systems (ICDCS'94), Poznan (Poland)*, pages 396–403. IEEE Computer Society Press.

[Kalyon, 2007] Kalyon, G. (2007). *Testing Distributed Systems Through Symbolic Model Checking: IDD, IST and BDD*. DEA Thesis, Université Libre de Bruxelles.

[Kalyon et al., 2007] Kalyon, G., Massart, T., Meuter, C., and Van Begin, L. (2007). Testing Distributed Systems Through Symbolic Model Checking. In *Proceeding of the 27thIFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'07), Tallinn (Estonia)*, volume 4574 of *Lecture Notes in Computer Science*, pages 263–279. Springer-Verlag.

[Knaster, 1928] Knaster, B. (1928). Un théorème sur les Fonctions d'Ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134.

[Kripke, 1963] Kripke, S. A. (1963). Semantical Consideration on Model Logic. *Acta Philisophica Fennica*, 16:83–94.

[Lamport, 1978] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565.

[Lamport, 1983] Lamport, L. (1983). What Good is Temporal Logic. In *Proceeding of the IFIP 9thWorld Computer Congress on Information Processing, Paris (France)*, pages 657–668. North Holland/IFFIP.

[LeGuernic et al., 1991] LeGuernic, P., Gautier, T., LeBorgne, M., and LeMaire, C. (1991). Programming Real Time Applications with SIGNAL. In *Proceedings of the IEEE*, volume 79, pages 1321–1336.

[Leuschel and Massart, 2000] Leuschel, M. and Massart, T. (2000). Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In *Proceeding of the 9thInternational Workshop on Logic Programming Synthesis and Transformation (LOPSTR'99), Veneza (Italy)*, volume 1817 of *Lecture Notes in Computer Science*, pages 62–81. Springer.

[Macq Electronique, 2006] Macq Electronique (2006). Programmable Logic Controller of the Fifth Generation. `http://www.macqel.be/images/movie/PLC-P5-en.pdf`.

[Markey, 2003] Markey, N. (2003). *Logique Temporelle Pour la Vérification : Expressivité, Complexité et Algorithmes.* PhD thesis, Université d'Orléans.

[Markey and Schnoebelen, 2003] Markey, N. and Schnoebelen, P. (2003). Model Checking a Path. In *Proceedings of the 14thInternational Conference on Concurency Theory (CONCUR'03), Marseille (France)*, volume 2761 of *Lecture Notes in Computer Science*, pages 248–262. Springer-Verlag.

[Massart, 1992] Massart, T. (1992). A Calculus to Define Correct Transformations of LOTOS Specifications. In *Proceeding of the 4thInternational Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'91), Sydney (Australia)*, pages 281–296.

[Massart et al., 2007] Massart, T., Van Begin, L., and Meuter, C. (2007). On the Complexity of Partial Order Trace Model Checking. Accepted for publication in *Information Processing Letters*, to appear.

[Mattern, 1989] Mattern, F. (1989). Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publisher B.V.

[Mazurkiewicz, 1987] Mazurkiewicz, A. W. (1987). Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 (part II)*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

[Meuter, 2003] Meuter, C. (2003). *Distribution of Reactive System*. DEA Thesis, Université Libre de Bruxelles.

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*. Springer.

[Milner, 1981] Milner, R. (1981). On Relating Synchrony and Asynchrony. Technical Report CSR-75-80, Computer Science Deptartement, Edinburgh University.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

[Møller et al., 1999] Møller, J., Lichtenberg, J., Andersen, H. R., and Hulgaard, H. (1999). Difference Decision Diagrams. In *Proceedings of the 8thAnnual Conference of the European Association for Computer Science Logic (CSL'99), Madrid (Spain)*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag.

[Monson-Haefel, 2001] Monson-Haefel, R. (2001). *Enterprise JavaBeans*. O'Reilly, 3$^{\text{rd}}$edition.

[Morin, 1999] Morin, R. (1999). Decompositions of Asynchronous Systems. In *Proceedings of the 9thInternational Conference on Concurrency Theory (CONCUR'98), Nice (France)*, volume 1466 of *Lecture Notes in Computer Science*, pages 171–178. Springer-Verlag.

[Myers, 1979] Myers, G. J. (1979). *The Art of Software Testing*. John Wiley and Sons.

[Nitzberg and Lo, 1991] Nitzberg, B. and Lo, V. (1991). Distributed Shared Memory: a Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60.

[Oestereich, 2002] Oestereich, B. (2002). *Developing Software with UML: Object-Oriented Analysis and Design in Practice*. Addison Wesley.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison Wesley.

[Peled et al., 2001] Peled, D., Valmari, A., and Kokkarinen, I. (2001). Relaxed Visibility Enhances Partial Order Reduction. *Formal Methods in System Design*, 19(3):275–289.

[Petri, 1962] Petri, C. A. (1962). *Kommunikation mit Automaten*. PhD thesis, Technical University Darmstadt.

[Pnueli, 1977] Pnueli, A. (1977). The Temporal Logic of Programs. In *Proceedings of the 18thAnnual Symposium on Foundations of Computer Science (FOCS'77), Providence (USA)*, pages 46–57. IEEE Computer Society.

[Priestley and Davey, 2002] Priestley, H. A. and Davey, B. A. (2002). *Introduction to Lattice and Order*. Cambridge University Press, 2ndedition.

[Queille and Sifakis, 1982] Queille, J.-P. and Sifakis, J. (1982). Specification and Verification of Concurrent Systems in CESAR. In *Proceeding of the 5thInternational Symposium on Programming, Torino (Italy)*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag.

[Raskin and Doyen, 2007] Raskin, J.-F. and Doyen, L. (2007). Improved Algorithms for the Automata-based Approach to Model Checking. In *Proceedings of the 13thInternational Conference on Tools and Algorithms for Construction and Analisys of Systems (TACAS'07), Braga (Portugal)*, volume 4424 of *Lecture Notes in Computer Science*, pages 451–465. Springer-Verlag.

[Rosu and Havelund, 2005] Rosu, G. and Havelund, K. (2005). Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.

[Schneider, 2004] Schneider, K. (2004). *Verification of Reactive Systems : Formal Methods and Algorithms*. Springer.

[Sen and Garg, 2003] Sen, A. and Garg, V. K. (2003). Detecting Temporal Logic Predicates in Distributed Programs using Computation Slicing. In *Proceedings of the 7thInternational Conference on Principles of Distributed Systems (OPODIS'03), La Martinique (France)*, volume 3144 of *Lecture Notes in Computer Sciences*, pages 171–183. Springer-Verlag.

[Sen et al., 2003] Sen, K., Rosu, G., and Agha, G. (2003). Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the 9thEuropean Software Engineering Conference (ESEC/FSE'03), Helsinki (Finland)*, pages 337–346. ACM Press.

[Sen et al., 2004a] Sen, K., Rosu, G., and Agha, G. (2004a). Online Efficient Predictive Safety Analysis of Multithreaded Programs. In *Proceedings of the 10thInternational Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), Barcelona (Spain)*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer.

[Sen et al., 2004b] Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2004b). Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceeding of the 26thInternational Conference on Software Engineering (ICSE'04), Edimburgh (United Kingdom)*, pages 418–427. IEEE Computer Society.

[Sen et al., 2006] Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2006). Decentralized Runtime Analysis of Multithreaded Applications. In *Proceedings of the 20thInternational Parallel and Distributed Processing Symposium, (IPDPS'06), Rhodes Island (Greece)*. IEEE Computer Society.

[Sistla and Clarke, 1985] Sistla, P. A. and Clarke, E. M. (1985). The Complexity of Propositional Linear Temporal Logic. *Journal of the ACM*, 32:733–749.

[Stefanescu et al., 1999] Stefanescu, A., Esparza, J., and Muscholl, A. (1999). Synthesis of Distributed Algorithms Using Asynchronous Automata. In *Proceedings of the 14thInternational Conference on Concurrency Theory (CONCUR'03), Marseille (France)*, volume 2761 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag.

[Stockmeyer and Meyer, 1973] Stockmeyer, L. J. and Meyer, A. R. (1973). Word Problems Requiring Exponential Time: Preliminary Report. In *Conference Records of*

the 5thAnnual ACM Symposium on Theory of Computing, Austin (USA), pages 1–9. ACM Press.

[Tanenbaum and van Steen, 2002] Tanenbaum, A. S. and van Steen, M. (2002). *Distributed Systems : Principles and Paradigms*. Prentice Hall.

[Tarjan, 1972] Tarjan, R. E. (1972). Depth First Dearch and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160.

[Tarski, 1955] Tarski, A. (1955). A Lattice-Theoretical Fixed Point Theorem and Its Applications. *Pacific Journal of Mathematics*, 2(5):285–309.

[Thiagarajan, 1994] Thiagarajan, P. S. (1994). A Trace Based Extension of Linear Time Temporal Logic. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS'04), Paris (France)*, pages 438–447. IEEE Computer Society.

[Thiagarajan and Walukiewicz, 2002] Thiagarajan, P. S. and Walukiewicz, I. (2002). An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. *Information and Computation*, 179(2):230–249.

[Turing, 1936] Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265.

[Valmari, 1993] Valmari, A. (1993). On-the-Fly Verification with Stubborn Sets. In *Proceedings of the 5thInternational Conference on Computer Aided Verification (CAV'03), Boulder (USA)*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag.

[Vardi, 1995] Vardi, M. (1995). Alternting Automata and Program Verification. In *Computer Science Today : Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag.

[Vardi and Wolper, 1986] Vardi, M. and Wolper, P. (1986). An Automata-Theoretic Approach to Automatic Program Verification. In *Proceeding of the 1stSymposium on Logics in Computer Science (LICS'86), Cambridge (USA)*, pages 332–334. IEEE Computer Society.

[Vardi, 2007] Vardi, M. Y. (2007). The Büchi Complementation Saga. In *Proceedings of the 24thAnnual Symposium on Theoretical Aspects of Computer Science*

(STACS'07), Aachen (Germany), volume 4393 of *Lecture Notes in Computer Science*, pages 12–22. Springer.

[von Bochmann, 1978] von Bochmann, G. (1978). Finite State Description of Communication Protocols. *Computer Networks*, 2:361–372.

[Walukiewicz, 2000] Walukiewicz, I. (2000). Model Checking CTL Properties of Pushdown Systems. In *Proceedings of the 20thConference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'00), New Delhi (India)*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138. Springer.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la Complexité*. Dunod, 2$^{\text{ème}}$ edition.

[Zampunieris and Le Charlier, 1995] Zampunieris, D. and Le Charlier, B. (1995). Efficient handling of large sets of tuples with sharing trees. In *Proceedings of the 5thIEEE Data Compression Conference (DCC'95), Snowbird (USA)*, page 428. IEEE Computer Society.

Index

- Symbols -

$(a, b]$ ... 11
2^X ... 12
$D \preceq D'$ ... 56
D_{max} ... 58
D_{min} ... 58
E_{X_i} ... 62
$F_{/p}$ ... 94
$K \models_{\mathsf{C}} \varphi$ ... 30
$K \models_{\mathsf{L}} \varphi$ ... 28
$K_1 \bowtie K_2$ ... 24
$K_1 \trianglelefteq K_2$ ... 24
$K_1 \trianglelefteq_S K_2$ ... 24
$K_1 \bowtie\!\!\!\!\sim K_2$ ... 25
$K_1 \trianglelefteq\!\!\!\sim K_2$ ... 25
$K_1 \trianglelefteq_S\!\!\!\sim K_2$ ... 25
K_T ... 95
$K_{P,D}^{\Theta,\Gamma}$ ... 83
$P \preceq P'$ ... 12
R^* ... 12
R^+ ... 12
R^i ... 12
R^{-1} ... 12
$T \times M$ ... 145
$T \otimes M$ ... 155
$T \models_{\mathsf{C}} \varphi$ ... 105
$T \models_{\mathsf{L}} \varphi$ ... 101
$T \models_{\mathsf{P}} \varphi$ ... 98
$V \models_{\mathsf{Q}} \varphi$ ... 20
$V \equiv V'$ ... 149
$V \models_{\mathsf{P}} \varphi$ ... 19
$X \mapsto Y$ ... 12, 13
$[a, b)$ ... 11
$[a, b]$ ... 11, 12
$\mathsf{A}(\varphi \,\mathsf{U}\, \psi)$ ... 29
$\mathsf{AF}\,\varphi$ ... 29
$\mathsf{AG}\,\varphi$ ... 29
$\mathsf{CC}(N, R)$ ... 126
$\mathsf{DC}_{\sqsubseteq}(X)$ ... 14
$\mathsf{E}(\varphi \,\mathsf{U}\, \psi)$ ... 29
$\mathsf{EG}\,\varphi$ ... 29
$\mathsf{EX}\,\varphi$ ... 29
$\mathsf{FP}(f)$ ... 16
$\mathsf{F}\,\varphi$ ... 27
$\mathsf{G}\,\varphi$ ... 27
$\mathsf{JI}(X)$ ... 16
$\mathsf{LB}_{\sqsubseteq}(Y)$ ... 13
$\mathsf{Max}_{\sqsubseteq}(X)$ ... 13
$\mathsf{Min}_{\sqsubseteq}(X)$ ... 13
$\Pi(X)$ ... 12
Σ^* ... 21
Σ^ω ... 21
$\mathsf{UB}_{\sqsubseteq}(Y)$ ... 13
$\mathsf{UC}_{\sqsubseteq}(X)$ ... 14
$\mathsf{X}\,\varphi$ ... 27
$\mathsf{after}(T)$ ... 178
$\mathsf{before}(T)$ ... 178
$\mathsf{body}(x)$ ... 53
$\mathbb{B}$ ... 11
$\bot$ ... 18
$\mathsf{Graphs}(T)$ ... 126
$\mathsf{cond}(e)$ ... 53

$\mathsf{crucial}(C, \varphi)$ . . . 123
$\downarrow X$ . . . 14
$\downarrow x$ . . . 14
$\diamond$ . . . 60
$\mathcal{D}_P$ . . . 55
$\mathsf{enabled}(C)$ . . . 96
$\mathsf{env}(T)$ . . . 178
$\mathsf{eval}[v](e)$ . . . 61
$\mathsf{evt}(x)$ . . . 53
$\widetilde{\mathsf{evt}}(x)$ . . . 53
$\exists p\, \varphi$ . . . 19
ff . . . 11
$\forall p\, \varphi$ . . . 19
$\mathsf{gfp}_{\sqsubseteq}(f)$ . . . 17
$\mathsf{glb}_{\sqsubseteq}(Y)$ . . . 13
$\mathsf{in}(s)$ . . . 21
$\mathsf{instr}(x)$ . . . 53, 54
$\mathbb{Z}$ . . . 11
$\mathcal{L}(\rho)$ . . . 24
$\lambda x \cdot f(x)$ . . . 13
$\mathsf{lang}(A)$ . . . 22
$|w|$ . . . 21
$\mathsf{lfp}_{\sqsubseteq}(f)$ . . . 17
$\mathsf{local}(s)$ . . . 53
$\mathsf{lub}_{\sqsubseteq}(Y)$ . . . 13
$\mathsf{NNF}(\varphi)$ . . . 141
$\mathsf{max}_{\sqsubseteq}(X)$ . . . 13
$\mathsf{min}_{\sqsubseteq}(X)$ . . . 13
$\mathbb{M}(k)$ . . . 12
$\mathbb{N}$ . . . 11
$\neg\varphi$ . . . 18
$s, e)$ . . . 149
$\nu \prec \langle \nu_1, \nu_2 \rangle$ . . . 71
ω-$\mathsf{lang}(A)$ . . . 22
$\omega \prec \langle \omega_1, \omega_2 \rangle$ . . . 71
ω-$\mathsf{runs}(S)$ . . . 23
ω-$\mathsf{runs}(q)$ . . . 23
ω-$\mathsf{traces}(K)$ . . . 24
$\mathsf{out}(s)$ . . . 21
$\overline{b}$ . . . 11
$\mathsf{path}(S)$ . . . 191
$\phi_{i \leftarrow j}$ . . . 61
$\mathsf{post}(X)$ . . . 23
$\mathsf{pre}(X)$ . . . 23
$\widetilde{\mathsf{pre}}(X)$ . . . 23
$S_{\searrow P}$ . . . 24
$\sigma_{\searrow P}$ . . . 24
$\mathsf{prop}(\varphi)$ . . . 18
$\mathsf{prop}(s)$ . . . 150
$\delta(e, b)$ . . . 94
$\mathsf{reach}(S)$ . . . 23
$\mathsf{runs}(S)$ . . . 23
$\mathsf{runs}(q)$ . . . 23
$\mathsf{in}(X)$ . . . 62
$[\![\varphi]\!]^T_{\mathsf{C}}$ . . . 105
$[\![\varphi]\!]^K_{\mathsf{C}}$ . . . 30
$[\![\varphi]\!]^{\mathsf{F}}_{\mathsf{L}}$ . . . 101
$[\![\varphi]\!]_{\mathsf{L}}$ . . . 28
$[\![\varphi]\!]^T_{\mathsf{P}}$ . . . 98
$[\![\varphi]\!]^T_{\mathbb{N}^k}$ . . . 177
$\mathsf{sensitive}(s)$ . . . 155
$\sigma \prec \langle \sigma_1, \sigma_2 \rangle$ . . . 72
$\sigma \simeq \sigma'$ . . . 26
$|S|$ . . . 23
$|T|$ . . . 94
$|\varphi|$ . . . 18
$\top$ . . . 18
$\mathsf{traces}(K)$ . . . 24
$\mathsf{treat}(X)$ . . . 62, 81
tt . . . 11
$\mathsf{tuple}(C)$ . . . 177
$\mathsf{tuple}C$ . . . 177

$\langle K, q\rangle \models_{\mathsf{C}} \varphi$ 30
$\langle N_1, O_1, s_1\rangle \sqsubseteq_{\otimes} \langle N_2, O_2, s_2\rangle$ 161
$\langle P, D\rangle \vdash q \hookrightarrow q'$ 62
$\langle T, C\rangle \models^{\mathsf{T}}_{\mathsf{C}} \varphi$ 105
$\langle \sigma, i\rangle \models_{\mathsf{L}} \varphi$ 27
$\langle \sigma, i\rangle \models^{\mathsf{F}}_{\mathsf{L}} \varphi$ 101
$\uparrow X$ 14
$\uparrow x$ 14
$\mathsf{out}(X)$ 62
$\mathsf{var}(x)$ 53, 54
ε 21
$\varphi \,\mathsf{U}\, \psi$ 27
$\varphi \Leftrightarrow \psi$ 18
$\varphi \Rightarrow \psi$ 18
$\varphi \wedge \psi$ 18
$\varphi \vee \psi$ 18
$\widetilde{\mathsf{var}}(x)$ 53
$\mathsf{width}_{\sqsubseteq}(X)$ 14
$e \preceq e'$ 94
$f[X \mapsto y]$ 12
$f[x \mapsto y]$ 12
$f \cup g$ 12
$f^i(x)$ 13
$q \overset{\sim}{\rightarrow} q'$ 25
$q \overset{\sim}{\hookrightarrow} q'$ 25
$q \rightarrow q'$ 23
$q \rightsquigarrow q'$ 23
$t < t'$ 12
$t \leq t'$ 12
$vc(x)$ 89
$w(i)$ 21
$w[i]$ 21
$w_1 \cdot w_2$ 21
$w_1 \parallel w_2$ 21
$w_{/\Sigma}$ 21
$x\, R\, y$ 12

`BCAST`(x,v) 60
`INPUT`(x) 60
`MSG` 60
`OUTPUT`(x) 60
`SEQ` 60

- A -

Acceptance testing 6
Alphabet 21
Alternating-bit protocol 164
Antichain 14
Antisymmetry 12
Arbitrary predicate 112

- B -

Bijective function 13
Birkhoff's representation 16
Bisimilarity 24
Black-box testing 6
Boolean logics 18
Boolean values 11
Büchi automaton 22

- C -

Chain 14
Channel distribution 71
Church's notation 13
Class events 41
    declaration 42
Classes 36
    declaration 38
Coarser distribution 56
Coarser partition 12
Code distribution 71
Communicating Finite State Machine . 75
    read transition 75
    semantics 76
    write transition 75

Communication channels 60
Complement 11
Complete lattice 14
Complexity
of CTL-TC 108
of CTL-TC for total order traces . . 108
of LTL-MC 28
of LTL-TC 104
of LTL-TC for total order traces . . . 102
of PBL-SAT 19
of CTL-MC 30
of PRED 99
of PRED for total order traces 100
of QBL-SAT 20
Composition 145
correctness 147
Computation graph 125
consistent cut 126
Computation of a slice 128
Computation slicing 125
Computational Tree Logic, CTL 29
model checking 30
semantics 30
semantics over po-traces 105
syntax 29
trace checking 105
Concatenation 21
Conjunction 18
Conjunctive normal form, CNF 18
Conjunctive predicate 120
Consistent cut 126
Construction of monitors 143
Continuous function 14
CORBA 3
Covering operator 161
monotonitcity 162
Crucial event 123
CTL trace checking problem, CTL-TC . 105
complexity 108
complexity for total order traces . 108
CTL-MC 30
Cut 95
Cuts
tuple representation 177

- D -

DCOM 3
Detection
of arbitrary predicates 112, 132
of conjunctive predicates 122
of disjunctive predicates 116, 137
of linear predicates 124
of local predicates 115
of observer-independent predicates 119
of regular predicate 131
of stable predicate 117
Determined monitor 149
Deterministic finite automaton 21
Dining philosophers 164
Disjunction 18
Disjunctive normal form, DNF . . . 18, 137
Disjunctive predicate 115
Distributed reactive control system 2
Distributed Supervision Language, dSL 35
classes 36
distribution 45
events 40
global variables 38
methods 39
sequences 44
sites 39
syntax 35
types 36

Distribution
coarser . . . 56
event-driven code . . . 45
finer . . . 56
of a dSL program . . . 55
sequential code . . . 47
Distributive lattice . . . 14
Downward closed set . . . 14
Downward closure . . . 14
dSL program . . . 53
distribution . . . 55
global state . . . 62
well-formedness . . . 55

- E -
EJB . . . 3
Envelope of a po-trace . . . 178
Equivalence . . . 18
Esterel . . . 3
Event-driven code . . . 40
distribution . . . 45
Events . . . 40
declaration . . . 42
modified treatment . . . 81
treatment . . . 62
Existential quantifier . . . 19
Explicit LTL trace checking . . . 147

- F -
Finally . . . 27, 29
Finer distribution . . . 56
Finer partition . . . 12
Finite automaton . . . 21
deterministic . . . 21
Fixed point . . . 16
Fully quantified formulae . . . 19

- G -
Global state . . . 95
Global state of a dSL program
definition . . . 62
Global variables . . . 38
declaration . . . 39
Globally . . . 27, 29
Greatest element . . . 13
Greatest fixed point . . . 17
Greatest lower bound . . . 13

- H -
Happened-Before relation . . . 90
Hasse diagram . . . 15

- I -
Implication . . . 18
Infix notation . . . 12
Injective function . . . 13
Input sampling . . . 62
Instrumentation . . . 7
Integer numbers . . . 11
Integration testing . . . 6
Interval Sharing Tree, IST . . . 191
path . . . 191
Intervals . . . 11
Inverse . . . 12
Isomorphism . . . 13

- J -
Join . . . 14
Join-irreducible elements . . . 16

- K -
Keywords
BOOL . . . 38
CLASS . . . 38
INPUT . . . 40

INT . . . 38
LAUNCH . . . 44
LONG . . . 38
METHOD . . . 41
OUTPUT . . . 40
REAL . . . 38
SEQUENCE . . . 44
SITE . . . 40
VAR . . . 39
WAIT . . . 44
WHEN . . . 42
WHEN IN . . . 42
Kripke structure . . . 23
trace . . . 24

- L -

Language . . . 21
Lattice . . . 14
Birkhoff's representation . . . 16
distributive . . . 14
join . . . 14
join-irreducible elements . . . 16
meet . . . 14
Tarski/Knaster iteration . . . 18
Least element . . . 13
Least fixed point . . . 17
Least upper bound . . . 13
Length . . . 21
Letter . . . 21
Linear predicate . . . 123
crucial event . . . 123
Linear Temporal Logic, LTL . . . 27
LTL-X . . . 28
model checking . . . 28
semantics . . . 27
semantics over finite sequences . . . 101
syntax . . . 27
trace checking . . . 101
Literals . . . 134
Liveness properties . . . 5
Local predicate . . . 114
Local state . . . 96
Local state of a process
communication channels . . . 60
Local state of a process . . . 60, 61
valuation . . . 60
workload . . . 60
Local state of a sequence
definition . . . 61
Lower bound . . . 13
LTL trace checking problem, LTL-TC . 101
complexity . . . 104
complexity for total order traces . 102
LTL-MC . . . 28
Lustre . . . 3

- M -

Maximal element . . . 13
Maximal run . . . 23
Meet . . . 14
Meet-closed predicate . . . 133
Methods . . . 39
call . . . 41
declaration . . . 41
Minimal element . . . 13
Model checking . . . 4
dSL programs . . . 51
modeling . . . 4
of CTL formulae . . . 30
of LTL formulae . . . 28
specification . . . 5
verification . . . 5
Modeling . . . 4
Monitor . . . 87, 140

construction . . . 143
determined . . . 149
Monitor driven composition . . . 155
correctness . . . 159
Monotonic function . . . 14
Multi-rectangles . . . 12

- N -

Natural numbers . . . 11
Negation . . . 18
Negation normal form, NNF . . . 18, 141
Next . . . 27, 29

- O -

Observer . . . 87
Observer-independent predicate . . . 118
Order ideal . . . 14
Output update . . . 62

- P -

Partial order . . . 13
Partial order trace, po-trace . . . 94
cut . . . 95
global state . . . 95
local state . . . 96
semantics . . . 95
Partially ordered set . . . 13
antichain . . . 14
chain . . . 14
downward closure . . . 14
greatest element . . . 13
greatest lower bound . . . 13
least element . . . 13
least upper bound . . . 13
lower bound . . . 13
maximal element . . . 13
minimal element . . . 13
order ideal . . . 14
upper bound . . . 13
upward closure . . . 14
Partition . . . 12
coarser . . . 12
finer . . . 12
Path of an IST . . . 191
PBL-SAT . . . 19
complexity . . . 19
Peterson mutual exclusion protocol . . . 164
Powerset . . . 12
Predicate detection problem, PRED . . . 98
complexity . . . 99
complexity for total order traces . 100
Programmable Logic Controller, PLC . 33
Projection . . . 21
Propositional Boolean Logic, PBL . . . 18
satisfiability . . . 19
semantics . . . 19
syntax . . . 18
Propositional sequence . . . 27

- Q -

QBL-SAT . . . 20
complexity . . . 20
Quantified Boolean Logic, QBL . . . 19
satisfiability . . . 20
semantics . . . 20
syntax . . . 19

- R -

Reachability problem
for CFSM . . . 76
Reactive system . . . 2
Read transition . . . 75
Reflexive and transitive closure . . . 12
Reflexivity . . . 12
Regular predicate . . . 125

Regular Temporal Predicate 168
Relation
antisymmetric 12
infix notation 12
inverse 12
reflexive 12
reflexive and transitive closure 12
total 12
transitive 12
transitive closure 12
Run 23
maximal 23

- S -

Safety properties 5
Satisfiability
of PBL formulae 19
of QBL formulae 20
Semantics
of CTL 30
of PBL 19
of QBL 20
of a dSL program 69
of a po-trace 95
of Linear Temporal Logic 27
Sensitive events 155
Sequences 44
call 44
declaration 44
local state 61
Sequential code 44
distribution 47
Shuffle product 21
Signal 3
Simulation relation 24
Sites 39
declaration 40
Size
of a transition system 23
Slice 127
Specification 5, 7
liveness properties 5
safety properties 5
Stable predicate 117
Strongly connected components 132
Stuttering bisimilarity 25
Stuttering equivalence 26
Stuttering simulation 25
Stuttering simulation relation 25
Stuttering transition 25
Supervision language, SL 33
Supervisor 33
Surjective function 13
Symbolic LTL trace checking 159
Syntax
of CTL 29
of dSL 35
of LTL 27
of PBL 18
of QBL 19
System testing 6

- T -

Tarski/Knaster iteration 18
Testing 5, 87
acceptance 6
black-box 6
instrumentation 7
integration 6
monitor 87
observer 87
specification 7
system 6
trace analisys 7

unit 6
white-box 6
Totality 12
Totally ordered set 13
Trace 24
Trace analisys 7
Transition system 22
run 23
size 23
Transitive closure 12
Transitivity 12
Tuple representation of cuts 177
Types 36

- **U** -

Undecidability
of CFSM-REACH 76
Unit testing 6
Universal quantifier 19
Until 27, 29
Upper bound 13
Upward closed set 14
Upward closure 14

- **V** -

Valuation 19, 60
Valuation distribution 71
Vector clock 89
mapping 89
message passing programs 90
shared variable program 92
Verification 5

- **W** -

Well-formed dSL program 53, 55
White-box testing 6
Word 21
concatenation 21
length 21
projection 21
shuffle product 21
Workload 60
Write transition 75